Imperial College
London

DEEP LEARNING CONTROL OF AN AUTONOMOUS ROBOT-TRAILER MATERIAL TRANSPORT SYSTEM

VLADIMIR KHON
Under supervision of Dr. Peter Chao Yu Chen

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF MECHANICAL ENGINEERING
APRIL 2020

Abstract

The rise in interest of Machine Learning over the last three decades has seen its growing application within all aspects of everyday technology and data analytics. From household devices, to autonomous vehicles, it is undeniable that Machine Learning is a driving force in the modern world. One specific aspect of Machine Learning that has seen a boost in fascination is Deep Learning in which a Neural Network is trained to detect representations of a dataset. This has extreme utility in image recognition, speech recognition, market value forecasting, recommender systems, and robotics.

This project aims at applying Deep Learning within the control of a robot-trailer material transport system for traversing a warehouse environment. After presenting the context, objectives, and approaches of the project, a review of the literature is conducted on the state of Deep Learning, followed by brief applications into robot path planning and perception and delving deeper into control. The robot-trailer system is described whilst presenting the Kinematics and Dynamics models for identifying the Neural Network inputs and outputs. The first objective is tackled which is learning the inverse of the models. Generation of the input/output based dataset is first described, then Recurrent Neural Networks are built with TensorFlow, and subsequently trained on personal and high-powered computing. Generated paths are analysed and it was found that Neural Networks with low Mean Squared Errors of 9.48×10^{-5} for Inverse Kinematics' velocities and 3×10^{-4} for Inverse Dynamics' torques were not enough to create collision-free paths but were effective on simpler paths.

Secondly, a Kinematic Controller was to be compensated for positional error. Approaches to solve this were explored, from implementing earlier TensorFlow networks, to manual Multilayer Perceptron design in Simulink. A Multilayer Perceptron was built in Matlab and an error of 166.1706 resulted, successfully improving over the Kinematic Controller error of 171.9726, and showing that Deep Learning error compensation is feasible. Insights from the project such as the limit to training complexity are realised whilst criticising the approach taken. It is suggested that more intelligent Neural Network layers and training methods should be used in future work. A few integrated control schemes are then presented for practical application within the context of the project. Suggestions for future work such as implementation of raw torque information or feedback control is also outlined for the advancement of the enabled robot-trailer control.

Acknowledgements

I would like to extend my sincerest gratitude to Dr. Peter Chao Yu Chen for his supervision on the project and support of my Deep Learning journey. My biggest thank you to Dr. Stelios Rigopoulos and Dr. Nicolas Cinosi for the opportunity to exchange at the National University of Singapore and the chance to pursue my passion for robotics and explore artificial intelligence.

All of my deepest affection towards my family for their relentless and unending love and support over the course of my life and time at university; no matter where we are, I am thankful that I can always reach out and rely on you. Thank you to my friends from Imperial College London; your encouragement, love, and laughs kept the days bright and memories vivid over the years. Thank you to my friends from NUS for their support throughout the year, especially the people at UTown Mac Commons whom have supported the development of this project. I would especially like to thank Eytan Merkier for his time in reviewing this report and for filling this year abroad with lots of laughs and memories.

Thank you, everyone.

Contents

List of Figures

List of Algorithms

1 Introduction

With recent popularity of Machine Learning, the development of intelligent robots has seen a significant boost. Tools such as Deep Learning have encouraged the development of smart control schemes that outperform conventional measures. The result is the creation of more autonomous systems that are able to traverse environments such as warehouses to deliver goods and materials without supervision. However, current systems need to be tightly controlled to reduce uncertainty,which increases design costs in retrofitting peripherals such as cameras, sensors, and beacons. It is hence more cost effective to deploy robots that attach themselves to current systems rather than have systems built to accommodate robots. One of these systems for transporting material uses a castor-wheeled trolley with complex kinematics and dynamics that the robot must correct for in motion. This brings new technical challenges to light. The two most important of these challenges is the generation of correct speeds and headings for creating correct trajectories followed by consistent trajectory following by the robot through the environments. This project subsequently aims at using Deep Learning to develop solutions to the aforementioned issues given the complex motion models that the system possess. The two subsequent objectives of this project are:

- To develop a Neural Network that can learn the Inverse Kinematics (IK) and Inverse Dynamics (ID) to generate linear and angular speeds as well as left and right wheel torques respectively,
- To combine conventional control approaches with a Neural Network (NN) to improve performance by reducing residual error.

The approaches taken were:

- The use of long- and short-term memory cells within Neural Networks,
- The use of an adaptive Multi-Layer Perceptron (MLP) for error reduction.

Before proceeding further, some terminology must first be defined as this will be used heavily in subsequent sections:

- Epoch - the iteration taken for the entire dataset to be presented to the network and for the network to be trained.
- Hyperparameter - any tunable or predefined parameter used for any aspect of training that is not a parameter of the neural network or dataset.
- Training - action of iteratively passing the dataset through the network, calculating error, and updating the weights accordingly.

2 Literature Review

2.1 State of the Art

To explore applications of Deep Learning within robot control, the definition of Deep Learning and its components must be uncovered. In essence, Deep Learning is a subset of Machine Learning within which linear/non-linear components (or cells) are employed to learn a given set of data for a supervised or unsupervised problem. Deep Learning is considered to be multi-level representation learning [1] that can recognise low-level patterns within the data and abstract away features to create a high level representation of a problem or function. The hallmark of Deep Learning is the Neural Network (NN) which is a set of layers of linear/non-linear cells to create a large, interconnected architecture that resembles the human brain. The simplest of these cells that can learn was introduced by Frank Rosenblatt in 1958 and is known as The Perceptron [3] which is modelled after the work done by McCulloch and Pitts [4]. The basis of the McCulloch and Pitts model is the human neuron[1] which was formalised (and heavily criticised to the point of demotivating all research) by Minsky and Papert in their book *Perceptrons* [5] as described in Equation (1) and is shown in Figure 1:

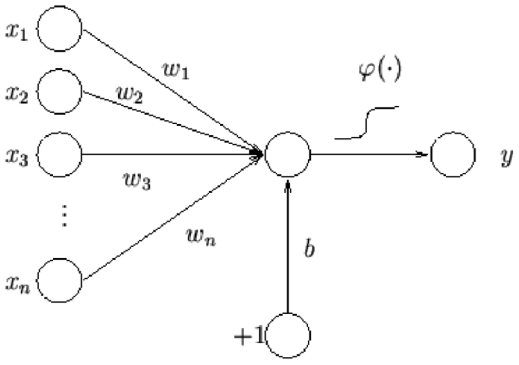


Figure 1: Perceptron [2].

$$y = \varphi(\sum_{i=1}^{n} w_i x_i + b) \tag{1}$$

where x_i is the ith input of an n sized input vector, w_i is the weight associated with the ith input, b is the bias of that neuron, φ is an activation function, and y is the output of the Perceptron [6]. Physically, the weight determines the 'strength of connection' to a specific input; the larger the weight, the stronger the connection. The bias is then added to shift the sum of inputs to remove the background shift within the dataset[2]. Summing all inputs and biases and passing through an activation function determines how the neuron 'fires'. A biological neuron fires when a threshold is exceeded which can be interpreted as a hard limiter. However, the advantages of artificial neurons is that the activation function can be changed to more suitably fit the problem.

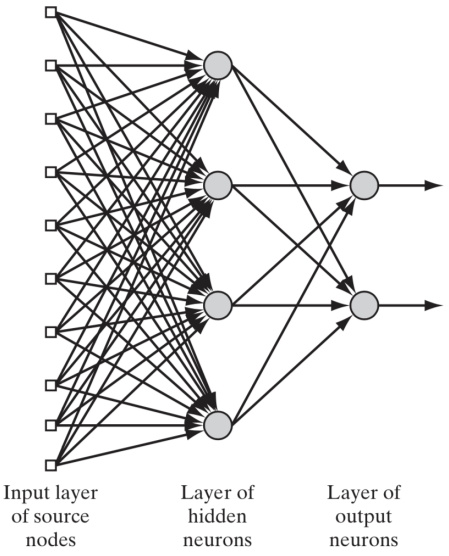


Figure 2: MLP [6].

[1] Nowadays, the cells of a Neural Network are colloquially referred to as neurons due to this reason.
[2] More technically, the bias is the intercept of a specific cell on the hyperplane of the output function.

The Perceptron is not limited to only behaving as one unit as multiple Perceptrons can be stacked together to form 'layers'. These layers can be stacked together to form a *deep* network of Perceptrons, also known as Multilayer Perceptrons (MLPs) as shown in Figure 2. Creating deeper MLPs gives them the ability to learn more complex datasets and functions, however the computational cost increases. Nowadays, the Perceptron and MLP have become too simple for most complex scenarios and hence more advanced cells and layers have been developed such as Recurrent cells and Convolutional layers. A form of Recurrent Neural Networks (RNNs) first originated from John Hopfield [7] when he addressed content-addressable memory and state-space of neurons. Convolutional Neural Networks (CNNs) are almost exclusively used in image recognition and were used early on in the 90's for facial identification by Lawrence et al. [8]. This was done by locally sampling images through a window that scans a region of pixels to create a representation vector of the pixel intensity and difference in intensities to the origin pixel.

Subsequently, these networks need to be 'trained' by updating the weights of each connection based on some learning criteria such as error reduction. Weight updating and training optimisers consequently have become a heavy topic of research within Machine Learning applications. Amongst all optimisers, the most groundbreaking was developed by Rumelhart et al. and is known as Backpropagation [9] (although invention is accredited to Seppo Linnainmaa [10] under the work of Reverse-Mode Differentiation). This weight update method uses a gradient descent based algorithm utilising the Delta Rule. The Delta Rule, simply put, is the product of the learning rate, gradient across an activation function, and the error between the desired and actual output. This is the update that is given to the weight at the output. The update is then propagated backwards to the previous layer's neurons across all connections to the outputs to update that layers input weights - *backpropagating* until the input layer is reached. This was seminal in taking NNs from theoretical research papers into tangible applications, however Rumelhart et al. stress the inherent limitations of the algorithm - the optimisation is bound to be trapped within local minima as with any gradient descent method.

Current optimisation methods approach the training from multiple independent and combined methods. However, almost all optimisers focus on two objectives: reducing a cost function (such as Mean/Sum Absolute Error (MAE/SAE) or Mean/Sum Squared Error (MSE/SSE)) and updating the weights. Notable optimisation methods include Bayesian Regularisation Backpropagation as worked on by Mackay [11] who demonstrated this method on a two-joint robot arm. It is a probabilistic technique that compares the posterior and prior probabilities when updating weights. These are proportional to cost function minimisation with the maximum likelihood near zero error. The weights are updated based on the Bayes Theorem of log-odds that minimise the cost function within Maximum Likelihood Learning. However, this can be computationally expensive for all weight calculations and so Bayesian Regularisation employs a partial-Bayesian technique to regularise the Maximum Likelihood.

Other methods such as Resilient Backpropagation (Rprop), Adaptive Gradient (AdaGrad), Root-Mean-Square Backpropagation (RMSProp), and Adaptive Moment Estimation (Adam) (which is based on RMSProp) have become more ubiquitous in current day Deep Learning [12].

Taking the latest addition to the field of optimisers, Adam, as developed by Kingma et al. [13], approaches training using momentum estimates from prior gradients (or, more accurately, exponentially decayed prior gradient average) for adaptive learning rates for the NN parameters. The update rule which Adam uses is shown in Equation (2):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2}$$

where θ is the network parameter, η is the learning rate, $\hat{m}_t$ is the first moment estimate, $\hat{v}_t$ is the second moment estimate, ϵ is the smoothing term (defaulted to 10^{-8}), and t is the current epoch. The first and second moment estimates are respectively updated by Equations (3) and (4):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4}$$

where the hyperparameters, β_1 and β_2, are the momentum decay parameter and scaling decay parameter which determine the rate of decay of the gradient and gradient squared averages. They are defaulted to 0.9 and 0.999 respectively. These averages are determined by Equations (5) and (6) with the gradient, g_t, and gradient squared, g_t^2, decayed average for the first, m_t, and second, v_t, moments:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{5}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{6}$$

Such adaptive learning rate measures allow for much more efficient and intelligent methods for optimisation and training whilst keeping computational expenses in memory and speed low. The above, for example, truly only keeps track of 3 parameters from the previous iteration which inherently carries information from all prior epochs without the need for storing all information from said epochs. As such, current Deep Learning projects can be done from the comfort of home on personal computers with scaling capabilities for increasing levels of seriousness.

Large-scale, commercial applications ultimately are the powerhouses of today's technology firms and fascination with intelligent machines. From image recognition networks such as AlexNet [14] and GoogLeNet [15] (Figure 3) to Natural Language Processing networks

such as Megatron-LM [16] and ALBERT [17], the development of large and complex networks brings artificial intelligence closer to replicating the human brain whilst deriving extensive use in all field such as autonomous vehicles [18], recognising speech of ALS patients [19], and a plethora more applications.

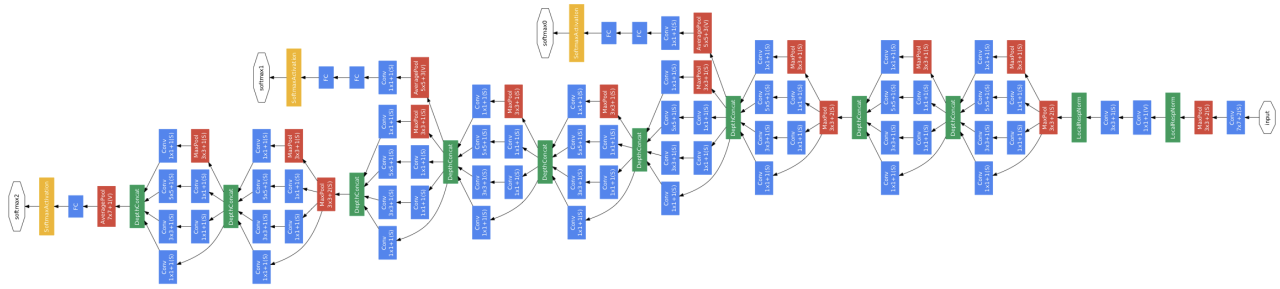


Figure 3: GoogLeNet [15].

As such, one of the most exciting points of application of Deep Learning is robotic systems, with Neural Networks manning aspects such as perception and control [20][21] to interact with the outside environment. These NNs tackle many challenges that come with perception and control such as high-dimensionality configuration space, unstructured environments, coordination, planning, sensing, data handling, high-frequency feedback, noise, etc.

2.2 Neural Networks and Robotics

The prior explained rise and development of Deep Learning within data-driven applications. This caused artificially intelligent machines to naturally spill over into robotics with research and development pushing towards fields such as perception, path planning, and control. An example of perception include SegNet [22], used to interpret objects by using a deep, VGG16 based CNN to encode and decode images to objects with a not fully-connected network. A path-planning example involves Unmanned Aerial Vehicle using Genetic Algorithms and NNs [23], with the genetic algorithm generating a set of correct paths for the NN to train on. However, as the scope of this project entails control, a review of current Deep Learning methods is conducted.

One restricting case for applicable robot control is the Kinematics and Dynamics of robotic systems as each system is unique and possesses configuration parameters such as number of actuators, number of static joints, or distances between joints which impact a robot's Degree of Freedom (DoF). Clearly, there is no unique solution that fits all robots and so the Kinematics and Dynamics of each system is uniquely derived. This led to boots-on-ground analytical approaches which served great use for control and trajectory generation [24], from using homogeneous transformation matrices to implementing Lagrangian mechanics. However, the forward models are easy to develop. It is with inverse mechanics where Deep Learning serves good purpose as the current solutions to IK and ID of robots

are numerical or analytical at best with problems of singularities, robot configuration, and over- and under-defined robot DoFs [25]. As such, Almusawi et al. developed an artificial NN to solve the IK problem of a 6 DoF robotic arm by applying a feedback loop to the system for error compensation and trajectory generation. Their proposed network proved to have a total error of 0.65% compared to 14.31% of a conventional, non-feedback NN which shows the capabilities to which NNs can learn IK problems. However, adding to the inverse mechanics problem complexity is learning of an ID model. This adds uncertainties such as internal friction, torque limits, current surges, elastic actuators, and change in configuration or robot design. All these factors play into how a robot, and especially robot manipulators, behave. Henceforth, Polydoros et al. [26] have developed a manipulator control NN with self-organising (General Hebbian Learning) decorrelator and reservoir (memory decaying RNN) layers that use Bayesian Inference for reservoir learning and real-time learning. The control diagram features a feedback controller to compensate for applied torques due to any unmodeled forces or torques within/on the manipulator. The results showed their training method and network had lower MSE loss and performed better on the Barret and BaxterRepeat datasets used by Local Weighted Regression Projection and Local Gaussian Process Regression training methods for real-time high dimensionality learning of robotic manipulators. It also has lower MSE loss per epoch than Sparce Spectrum Gaussian Process learning and the previous 2 learning methods. These two architectures for IK and ID learning both possess trajectory inputs (with IK involving orientation) and their respective outputs of joint angles and torques. It is hence viable for Deep Learning to learn the IK and ID and perhaps conduct open-loop control of robots based on well-developed NN architectures and compensation of unexpected errors.

A major issue that robotics faces today is an adequate solution to physical constraints and response of mobile robots. A method of overcoming physical constraints is presented by Chen et al. [27] via an Inequality Equality Constrained Optimisation Recurrent Neural Network that uses Lagrangian criteria optimisation, position values, and constraint inequalities for a time-variant error vector as the input to the RNN which outputs joint speeds. Such an approach considers the robots limitations into the constrained minimisation problem involving Lagrangian multiplication for inequalities within the cost function. Fuzzy logic has also been wed with NNs by Wang et al. [28] whose Deep NN comprises of membership layers containing membership activation functions, fuzzy rules layers which calculate applicability based on antecedent fuzzy rules, and a normalisation layer. The outputs are then passed through a defuzzification process to output the angle of disturbance to the robot for course correction. The training within itself only uses 3 cardinal and 2 ordinal directions biased to the front of the robot sensor.

The above pave the way for incorporating Deep Learning within IK and ID learning and intelligent error compensation for robotic control systems. Consequently, the project's system and configuration must be defined.

2.3 Task Description

The project subsequently focuses on developing control from the following robot-trailer configuration as seen in Figure 4:

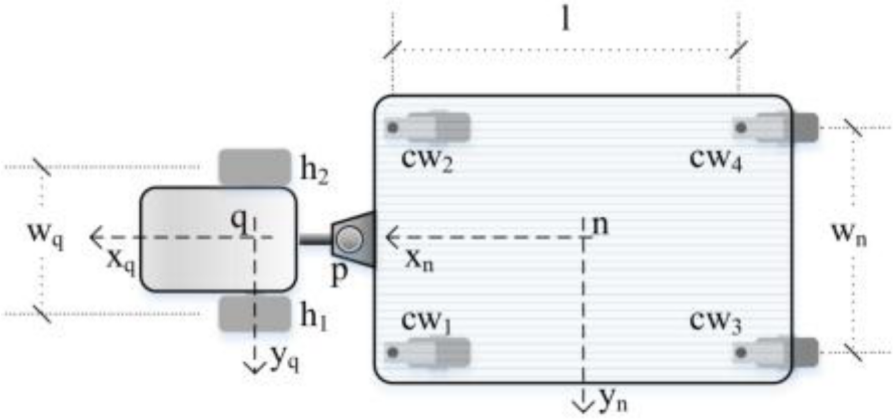


Figure 4: Robot-Trailer configuration [29].

The system comprises of a 4-castor-wheeled trailer, n frame origin, attached by a pin joint, p, to differential-drive robot, q frame origin. The robot drives the system and controls the trailer via driven left and right wheels, h_1 and h_2 respectively. The robot parameters are described in roboparam.m of Appendix A, however the above parameters of w_q is 0.387m, l is 0.920m, and w_n is 0.480m. Subsequently, the Kinematic model of the system is described by Equation (7):

$$\dot{\xi} = \begin{bmatrix} \dot{x_n} \\ \dot{y_n} \\ \dot{\theta_n} \\ \dot{\theta_q} \end{bmatrix} = \begin{bmatrix} cos(\theta_n)cos(\theta_n - \theta_q) & -\rho_1 cos(\theta_n)sin(\theta_n - \theta_q) \\ sin(\theta_n)cos(\theta_n - \theta_q) & -\rho_1 sin(\theta_n)sin(\theta_n - \theta_q) \\ -sin(\theta_n - \theta_q)\frac{1}{\rho_2} & -cos(\theta_n - \theta_q)\frac{\rho_1}{\rho_2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_q \\ \omega_q \end{bmatrix} \tag{7}$$

where $\dot{\xi}$ is the derivative of the ξ vector of x_n and y_n trailer positions, θ_n trailer orientation, and θ_q robot orientation with ρ_1 = 0.5m and ρ_2 = 1m as the distances of the robot centre to joint and joint to trailer centre respectively, and v_q and ω_q as the linear and angular speeds respectively. The model above describes how the motion of the robot is translated into the motion of the trailer. More importantly, it provides the input (I) and output (O) variables that are needed to find the IK of the system. These inputs/outputs (I/Os) are essential for defining the IK model and are found to be the vector ξ of trajectory and orientation information and the vector $[v_q\, \omega_q]$ of the robot's linear and angular speed. The Dynamic model of this system was also considered and is shown in Equation (8) below:

$$M(\bar{\xi})\ddot{\bar{\xi}} + n(\bar{\xi})\dot{\bar{\xi}} + \tau_d = B(\bar{\xi})\tau - \bar{L}_{\bar{\xi}}\lambda \tag{8}$$

where $\bar{\xi}$ is the vector $[\xi^T\ \dot{\phi_{h1}}\ \dot{\phi_{h2}}]^T$ with $\dot{\xi}$ and $\ddot{\xi}$ being the first and second derivatives respectively where $\dot{\phi_{h1}}$ and $\dot{\phi_{h2}}$ are the angular speeds of the robot's left and right wheels, $M(\bar{\xi})$ is the square positive definite inertia matrix, $n(\bar{\xi})$ is the centripetal and Coriolis matrix,

τ_d is the unmodeled dynamics and disturbances, $B(\bar{\xi})$ is the input transformation matrix, τ is the input torques, $[\tau_1 \ \tau_2]^T$, for the left and right wheels respectively, $\bar{L}_{\bar{\xi}}$ is the system's kinematic constraints and λ is the Lagrangian multiplier. Clearly, this heightened level of complexity of the robot-trailer system Dynamics causes issues for conventional control with singularities and infeasible solutions due to large torques and configurations. However, the I/Os selected from this system are the trajectory variables, ξ, defined earlier and the torques, τ. The augmented trajectory variables $\bar{\xi}$ were not used as the NN was to be trained based on the final physical configurations whilst remaining consistent between the IK and ID model inputs. The angular positions of the wheels are not part of the robot-trailer system position and orientation description but configure it. Note that the above forward mechanics are only used for simulation purposes to generate the training datasets using built in differential equation solvers. After considering the above task and configuration, the networks for IK and ID models were developed for Deep Learning.

3 Inverse Model Learning

Given the inputs and output derived earlier, an NN can be created based on the models considered. The first step is to consider the data and the means of generating a dataset. The Inverse Kinematic model is then considered for a proof-of-concept for generating adequate trajectories on simple environments. More complex cases are subsequently explored to test the ability of handling complex paths. Inverse Dynamics is then indulged for possible applications and training feasibility.

3.1 Dataset

The datasets for finding the inverse models were generated in Matlab and Simulink. A script was made to simulate and iterate through different environments in a random order (see Appendix B for code). three environment shapes were selected: an S-shape, a U-shape, and an SNS-shape. The first two serve as generic components to learn whilst the last is a combination of the first two to train a more complex case. Each sample comprises of 3200 time-steps which represent a time of 80 seconds at 0.025 second intervals. This was a balance between the stability of the Runge-Kutta numerical solver (ode4) of Simulink and the speed at which training can occur. For a larger number of time-steps, the solver gives more precise solutions without producing infinite gradient errors, however the time taken per epoch is sacrificed and training speed is decreased as more time-steps and features are considered. These features are segmented into inputs and outputs for the IK and ID models:

- X-position (I)
- Y-position (I)
- Trolley orientation (I)
- Robot orientation (I)
- Linear Speed (O) - IK
- Angular Speed (O) - IK
- Left Wheel Torque (O) - ID
- Right Wheel Torque (O) - ID

The above features are derived from the Kinematic Model and Dynamic Model presented earlier. The input can be seen as a 3200 by n by 4 cube and the output is a 3200 by n by 2 cube where n is the adjustable dataset size. This is subsequently shown below in Figure 5 are the dataset samples that were presented to the network:

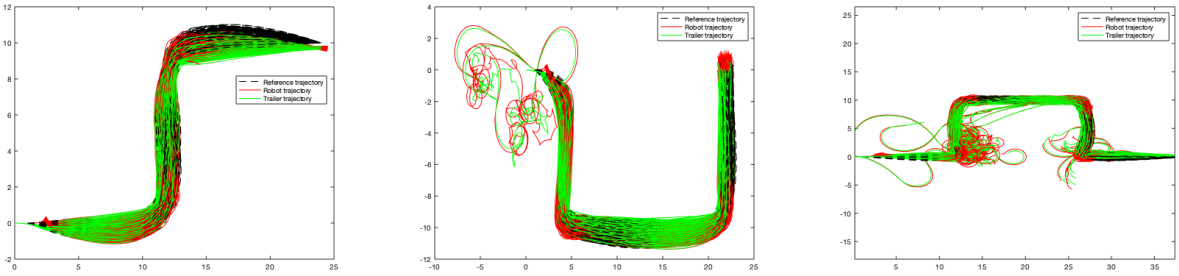


Figure 5: S-shaped, U-shaped, and SNS-shaped path dataset (y vs x coordinates).

The dataset above contains cases with high deviation from the route as seen by the haphazard paths. These cases were removed from the dataset by thresholding their MSE loss at 150. Figure 6 shows the MSE vs. Sample Number across the whole dataset, with large spikes corresponding to bad test runs. This threshold was determined experimentally by observing multiple runs of the scripts. A final dataset size of 600 (88.7MB) was generated for the IK model (Appendix B.A) and a larger size of 1000 (142.2MB) for the ID model (Appendix B.B) was created such that the NN is able to more effectively perform. The relatively unpredictable and noisy nature of torque data increases the difficulty of the NN to predict adequate torques. Incorporating a larger dataset gives more samples and hence increases likelihood of learning patterns. As a final step, the dataset is then finally segmented into 70% training, 20% validation, and 10% testing data before presenting the dataset to the NN.

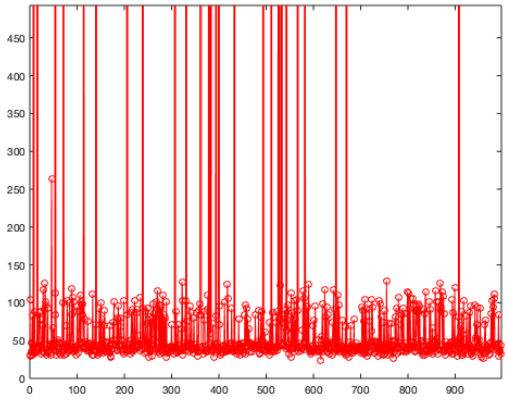


Figure 6: MSE loss across all data.

3.2 Neural Network Characteristics

To ensure that the NN can interpret the time-variant data, a memory based network must be devised to remember previous speeds and velocities. This is essential for the network as it must learn the effect of velocities from prior time-steps. This requires cells to have weighted information allocated to memory for the next time steps. Such a network is the RNN described earlier and is shown in Figure 7 for a single layer. The input vector, $\mathbf{x}$, is sequentially presented to the layer for n time-steps. The input vector is multiplied by a weight vector, $\mathbf{w}_x$, for the cell input and an output vector, $\mathbf{y}$, and a hidden state, $\mathbf{h}$, are passed on to the next layer and next time-step respectively. The hidden state, $\mathbf{w}_y.\mathbf{y}$, is then passed over to the next time-step. This gives the NN the ability to roll over patterns and data from previous time-steps and learn the time-variant patterns.

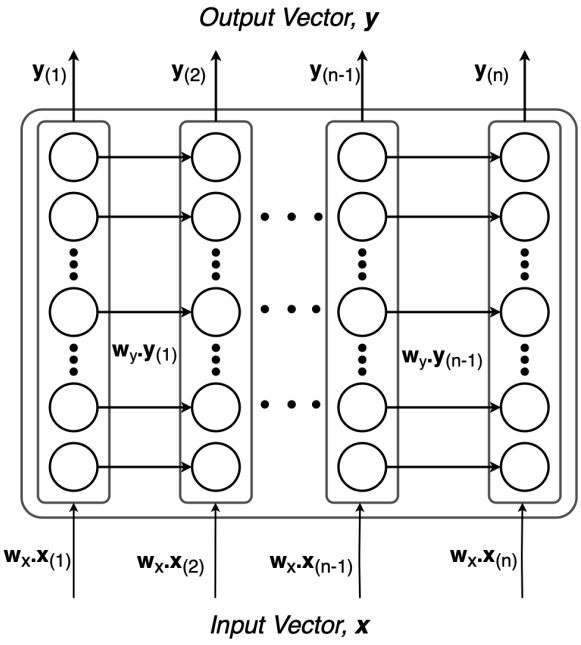


Figure 7: Recurrent Layer.

A specialised cell type that was extensively used in this project is the Long-Short Term Memory (LSTM) cell (shown in Figure 8) which can learn patterns over much longer time spans. The difference between a normal Recurrent cell and the LSTM cell is that the output of the cell to the subsequent time-step has 2 states: the short term state, $\mathbf{h}_{(t)}$, and the long

term state, $\mathbf{c}_{(t)}$ [30]. The cell uses the short term state to determine which data to forget in the long term state. The cell consists of 4 sub-cells, 3 with a logistic sigmoid function (logsig) with a range of 0 to 1 and 1 with a hyperbolic tangent sigmoid function (tansig) with a range of 1 to -1. The logsig cells, $\mathbf{f}_{(t)}$ and $\mathbf{i}_{(t)}$ outputs, act as gates to determine which information to forget and which to remember within the long term state. The tansig gate, $\mathbf{g}_{(t)}$, outputs the normal weighted information from the inputs, $\mathbf{x}_{(t)}$, which is added to the long term state. The output gate, $\mathbf{o}_{(t)}$, determines which information to output and put into the short term state. In addition to these advantages, LSTM cells can be set to be bidirectional which also reverses presentation of data from the last time-step to the first time-step which boost information selection into the respective states.

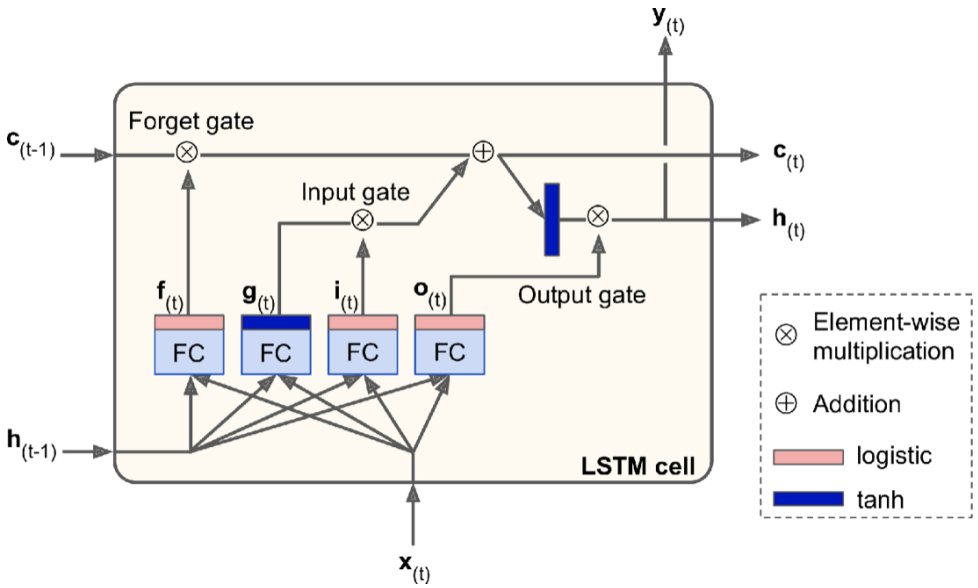


Figure 8: Long-Short Term Memory Cell [30].

The LSTM cell has clear advantages of learning time-variant patterns in data due to passing short and long term memory states. However, a similar type of cell, more specifically a type of layer, that is heavily used within image recognition problems is the 2D Convolution Layer which forms CNNs described earlier. Within image recognition, the cell is connected to pixels within a specific region of the image known as a receptive field, or *kernel*, of size $i \times i$ pixels. The distance between the regions covered by adjacent cells is the *stride*. Outputs of the cells are derived by scanning each region and going through the strides. The outputs of the layer are then multiplied by a matrix known as a *filter* that extracts specific features (e.g. a matrix to extract horizontal/vertical lines or origins of a kernel). This logic can be applied to time-variant, sequential data through a 1D Convolution Layer. Application of this as the input layer of the NN is explored in this project.

As the LSTM cell requires a heightened level of data handling than Perceptron cells, the right software and hardware must be used to process the memory. Python was used to leverage the extensive Machine Learning libraries available. TensorFlow's Keras API (tf.keras) was selected as the Deep Learning tool due to the exhaustive selection of cell types (such

as LSTM and its variants) and ease of implementation. Along with automatic parameter selection and diverse range of optimisers, fine tuning can be conducted and methods such as learning rate scheduling can be used to optimise the network further. Keras utilises 2 distinct network architectures being Sequential and Functional. This project specifically had relied heavily on tf.keras.models.Sequential() method which creates a full feedforward NN.

The hardware used in the beginning phases was a MacBook Pro 2.3 GHz Quad-Core Intel Core i7 for simple Inverse Kinematics models and small datasets (100 samples). As the complexity of the problem and size of the dataset increased, the training was then migrated to High Performance Computing (HPC) at NUS which possess a multitude of CPUs and GPUs for processing. The HPC clusters are the Volta processing nodes running on NVIDIA Tesla V100 GPUs with throughput speeds 32x faster than typical CPUs [31]. This reduces the average time for training on complex NNs (Dynamics) and large datasets (1000 samples) from 300 seconds per epoch down to 20 seconds per epoch (a 15x increase in training speed). The drawbacks with using HPC computing, however, is the wall-time of 72 hours and unknown queue times. Any jobs running for longer than the wall time are terminated automatically and future jobs are executed.

3.3 Inverse Kinematics

The initial IK RNN architecture used was 4LSTM-10LSTM-7LSTM-2Dense with the Dense layer set on Time-Distributed mode to output the linear and angular speeds at every time-step of the sequence. The selected optimiser was Adam due to its stability in handling sudden gradient shifts and dataset noise when training. A static learning rate of 0.01 was also applied. To ensure that a NN can be used to generate a set of paths, the Neural Network was made to learn only the S-shaped samples. A small dataset of 100 samples was generated within legacy code using simple velocity scheduling (Appendix C) with randomised fluctuations to embellish the dataset. This gave the results of Figure 9 below for two runs:

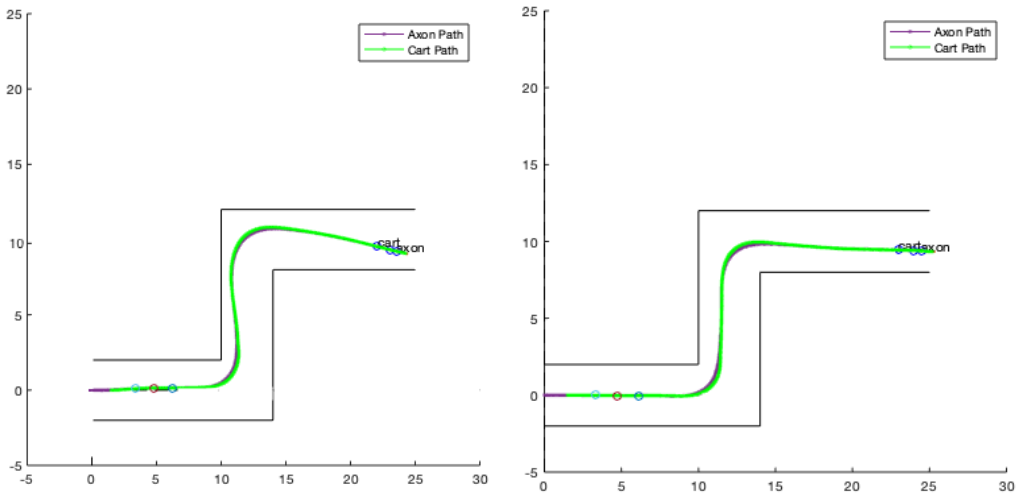


Figure 9: Neural Network S-shaped results (y vs x coordinates).

Figure 9 was the result of 1000 epochs at a dataset size of 100. The training took 1.75 hours (6.3 seconds per epoch) on the i7 processor. This confirms the NN's ability in learning a set path and following a trajectory by outputting linear and angular speeds. The scale was then taken up to a dataset size of 300 with the 3 shapes mentioned above. The RNN was redesigned with bidirectional LSTMs (biLSTMs) into a 4biLSTM-10biLSTM-7biLSTM-2Dense network and trained for 1200 epochs to produce the results of Figures 10 and 11.

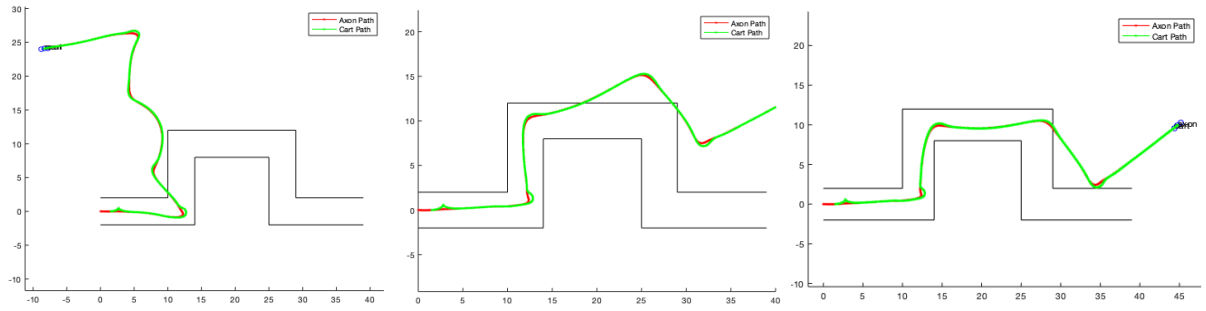


Figure 10: NN results for 300 samples with 500, 900, and 1200 epochs.

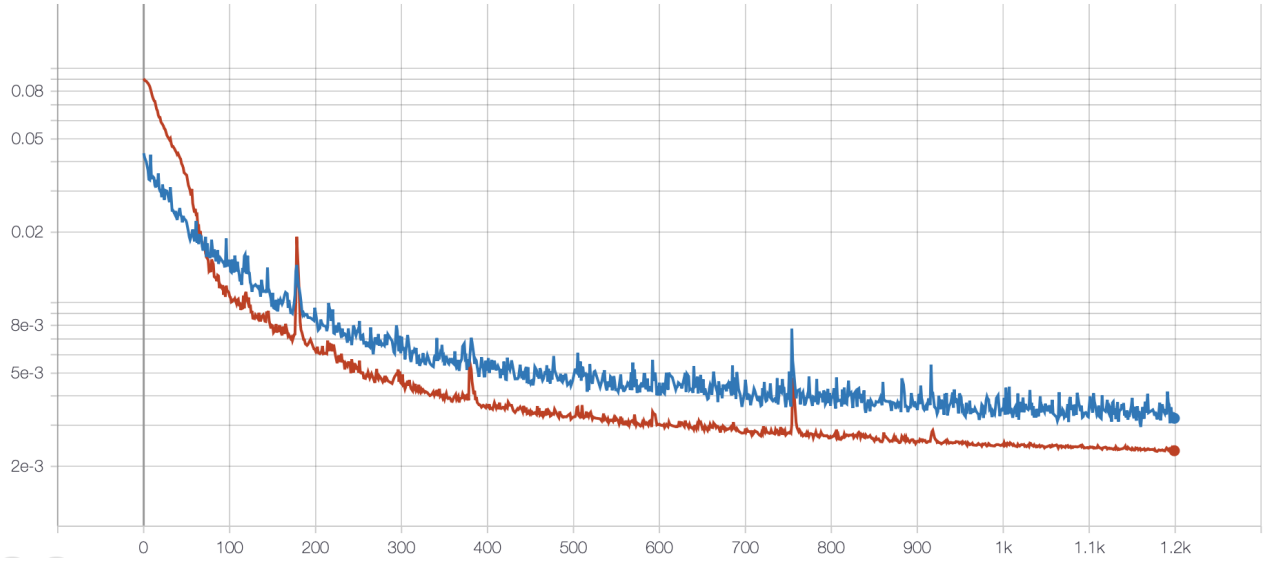


Figure 11: MSE loss vs epochs for IK model.

The above training was done over 2.1 hours (6.3 seconds per epoch) on the V100. The results promise better improvements with longer training which is reinforced by the MSE loss vs epochs of Figure 11 for both the training (blue) and validation (burgundy) data (acquired from TensorFlow's TensorBoard live visualisation tool). However, further testing has concluded that there are limited improvements with longer training as the MSE loss tends to take exponentially longer to half in value. For epochs up to 30000 the MSE only marginally decreases and still proves to have sub-optimal performance (discussed later) with training time taking 52.5 hours. Exponential learning rate scheduling was applied to slow down updates to the weights. This scheduling is shown below in Equation (9) below:

$$\eta = 0.1^{\frac{s}{epochs}} \eta_i \tag{9}$$

where η is the current learning rate, η_i is the initial learning rate, and s is the current epoch. This allows the training to inch closer to the minimum and avoid overshooting, causing the MSE loss to drop more substantially for the same time-frame whilst extracting more performance out of the RNN. The final performance for 8000 epochs is shown in Figure 12 for two separately trained Neural Networks acquired after 14 hours of training:

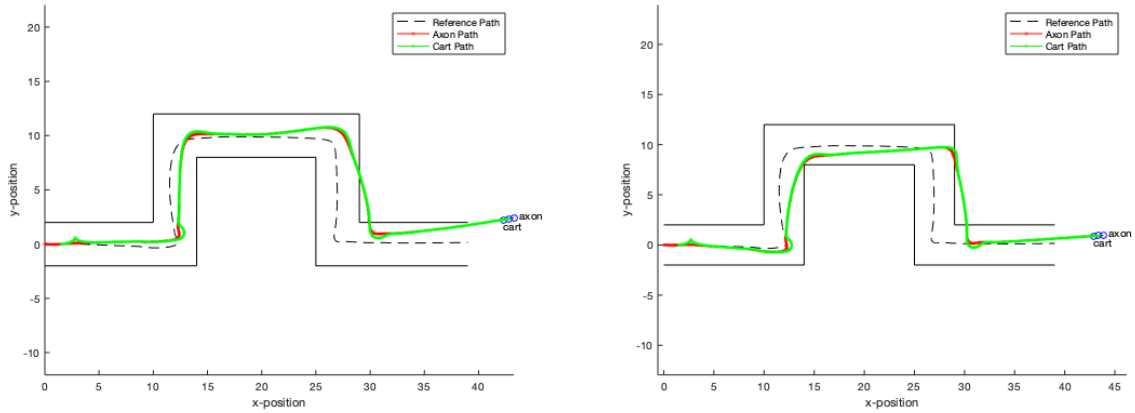


Figure 12: Neural Network S-shaped result (y vs x coordinates).

The path generated on this environment shape is acceptable for the first half of the path but deteriorates for the last half. There is clear collision with the environment borders which is clearly not desired along with increasing error from the desired reference path. The complexity of the path can be too much for the NN to learn and hence combining the S-shaped and U-shaped path to support training on the SNS-shaped path proves to be difficult. Regardless, the first half and deployment on the S-shape has revealed the maximum level of complexity to which the LSTM RNN can learn. From such results, it is then possible to postulate various control algorithms to manage the NN which will be discussed later.

After running several test runs with diminishing improvement, a Convolution Input layer (Conv1D) was added to exploit the pattern recognition abilities of such layers. The Conv1D layer is able to learn a specific pattern of the trajectory within a time-span segment of the dataset which can be passed to higher levels of the LSTM network. Combined with LSTM layers, the NN can recognise which trajectory patterns existed to predict the next linear and angular speeds by combining past and future trajectory patterns. The aim is give the NN the ability to learn components such as bends and straights. This gave an architecture of 4Conv1D-20biLSTM-10biLSTM-7biLSTM-2Dense with the Conv1D layer possessing 20 filters, 20 kernel size, and 1 stride to keep the same time-distributed output shape. The training was run for 12000 epochs for 19.25 hours to give the trajectories of Figure 13 on the following page with the MSE loss and learning rate vs. epochs.

The performance is similar to that of a typical LSTM RNN Network taking the same training time. The difference is the Conv1D layer creates more sudden spikes in MSE loss due to training attempting to escape local minima hence the minimisation occurs rapidly but stochastically unlike previous attempts which minimised gradually. The stochastic nature is advantageous to escape MSE loss stagnation whilst training at a faster rate.

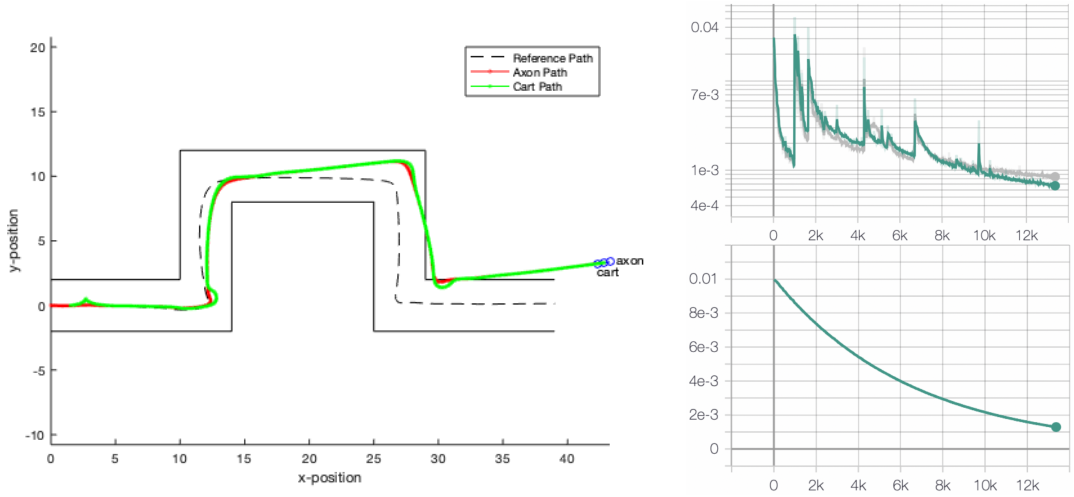


Figure 13: Convolutional Input Layer results after 12000 epochs.

Combining Convolutional and LSTM layers further, the architecture was changed to 4Conv1D-50biLSTM-30biLSTM-10biLSTM-2Dense with a kernel size of 320 in an attempt to further extract performance from the IK model. The increased kernel size is used in the hope of convolutional neurons generalising larger, complex patterns of the sequential data. The MSE loss plot is shown in Figure 14 below:

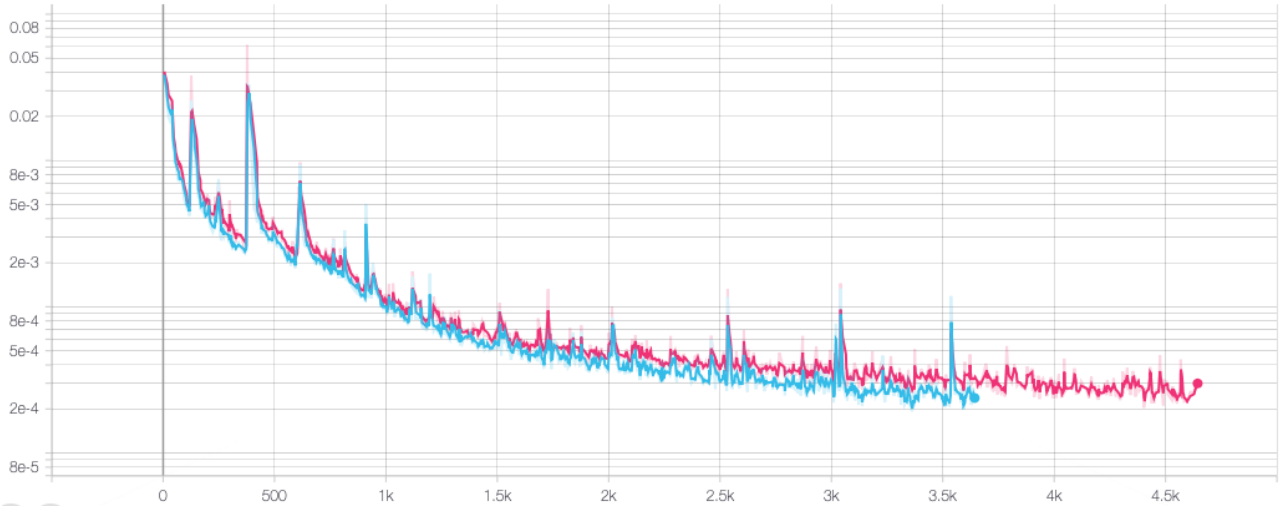


Figure 14: MSE loss for CRNN with kernel size of 320.

There is a clear increase in performance as the MSE loss now reaches to an average of 2.7 x 10^{-4} at 3500 epochs after 14.7 hours training. This increase is predominantly due to the increased number of LSTM neurons and layers within the architecture of the NN. The Conv1D layer does not impact the decrease in MSE but allows the network to avoid local minima early within the training. However, the increased kernel size no longer spikes the MSE loss due to stochastic local minima escapes. The impact of the Conv1D was bluntly removed when comparing against a pure LSTM RNN as described in the following section.

3.4 Results - Inverse Kinematics

After exploring both pure LSTM layers and Convolutional layers, the final architecture selected was 4biLSTM-50biLSTM-30biLSTM-10biLSTM-2Dense (see Appendix D for Python Code). This architecture was selected to ensure underfitting does not occur whilst following the observed trend of better performance with increasing neuron count as seen from IK and ID model learning. It is similar to the final Convolutional RNN architecture but removes the Conv1D layer to avoid MSE loss spikes in escaping local minima. The Conv1D layer also does not improve performance over long time frames and was deemed to be redundant. The IK RNN results are shown in Figure 15:

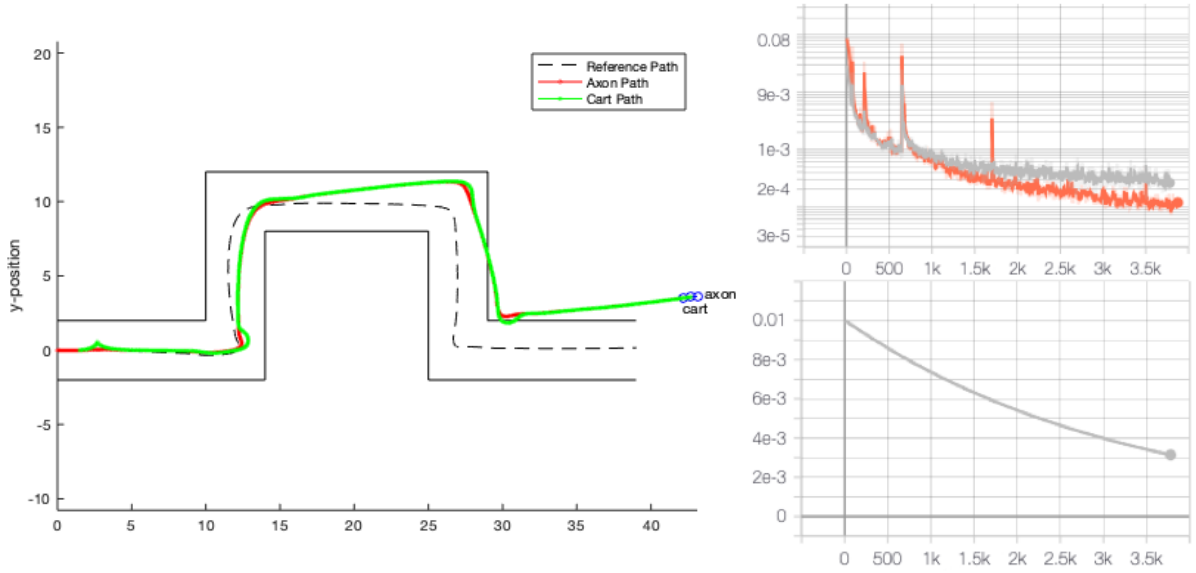


Figure 15: Final IK result (left) with MSE loss (top)/Learning Rate (bottom) vs. epochs.

The above contains the least MSE of 9.48 x 10^{-5} on the test data within 3500 epochs over 15.5 hours (15.3 seconds per epoch). This is significantly faster than using the Convolutional RNN which requires 12000 epochs to reach and MSE loss of 1 x 10^{-3}. However, given the above MSE losses, the actual change in MSE loss becomes exponentially smaller for the same time trained. The increase in performance then becomes marginal and there is very slight (but exponentially diminishing) improvement of the path following of the robot-trailer system for the complex case shown above. The trailer will always be dangerously close to the borders of the environment even when the physical dimensions of the system are not considered. This is not acceptable as a risk of material damage is presented. It is hence suggested to scale back the complexity of the environment for the NN and focus only on the lowest common denominator components such as S-shapes or corners.

3.5 Inverse Dynamics

In tandem with the IK training, ID training was conducted to test the ability of RNNs in learning the inverse dynamics of the complex castor wheel system and to increase the complexity

by training on the 1000 sample dataset. The NN architecture was a 4biLSTM-50biLSTM-30biLSTM-10biLSTM-2Dense network which is equal to that of the IK RNN due to the increased complexity of the dataset which can cause underfitting (see Appendix D for Code). The training was done similarly to the IK RNN on the HPC for faster performance as the time taken to train on generic CPUs requires 83.3 hours for 1000 epochs for any training with low MSE loss to be conducted. The results for 8000 epochs after 42.89 hours are shown with the MSE loss and learning rate vs. epochs in Figure 16 below:

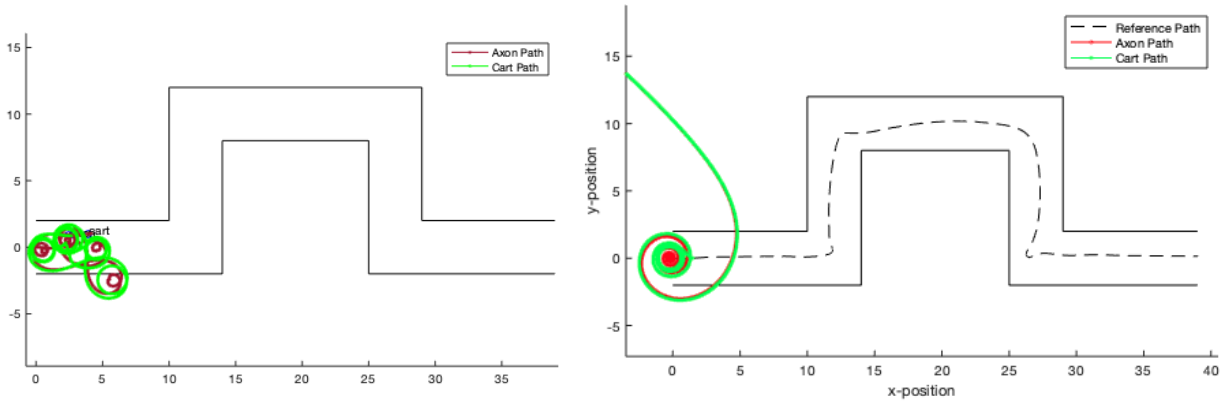


Figure 16: RNN result for complex case, older run (left) and newer run (right).

It is clear that the Inverse Dynamics are too complex for the Neural Network to handle due to high levels of noise and variability within the left and right wheel torque data. The large variance in the dataset due to 3 different environments also poses a difficulty to the RNN as constant weight adjustment within random data presentation is done to account for the change in input variable. Retraining the ID NN on pre-loaded weights from earlier runs was also attempted however the MSE loss did not significantly decrease and stagnation occurred. Additionally, resetting the learning rate schedule when retraining causes significant perturbations in the beginning stages due to spikes in MSE loss which delays training to optimal conditions but allows the NN to escape local minima. However for a case with complex dynamics, there is uncertainty whether a global minimum can be effectively and efficiently reached. The learning rate reset effect and stagnation of training is seen in Figure 17 below for 2 separate pre-loaded weights (first for 8000 epochs and then for 6000 epochs):

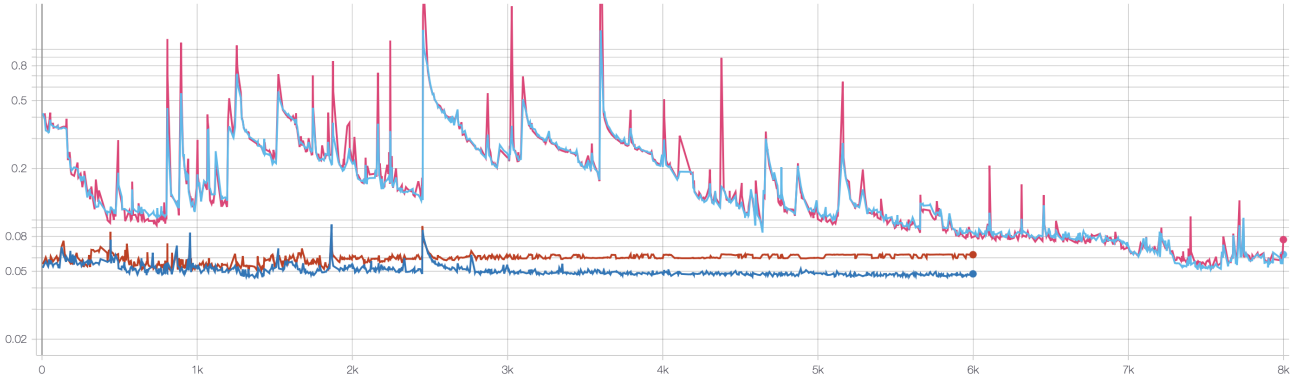


Figure 17: ID training MSE loss vs. epochs plot, upper lines for first run and lower lines for second run.

It can be seen that the algorithm struggles to train the RNN on the ID model from the perturbations and stagnation of the first run and the second run respectively. In a majority of cases, the jump in MSE is a good indicator of the optimiser escaping local minima whilst, in this case, the desired behaviour is a steady decrease indicating approach to optimum. The time taken for training to stabilise and the MSE to steadily decrease expends computational time and resources whilst running the risk of stagnating the MSE loss. This can be seen from the lower two lines of validation and training at which longer training increases the MSE loss on the brown line (validation). The training does not decrease the MSE loss below an adequate threshold and stagnates at a value far from ideal. Granted that the datasets differ for the IK and ID models, with the velocity and torque outputs possessing different units, it was learnt from the IK RNN training that the target MSE loss lies within 1 x 10^{-5}. As the order of magnitudes for the IK and ID datasets vary by a factor of 10, it can be assumed that the target MSE loss for the ID model lies below 1 x 10^{-4}. From the above graphs, the variance in the dataset disables ideal and steady training and hence the average MSE loss of the training runs will lie at a magnitude of 1 x 10^{-2}, 100x more than the target MSE loss.

The complexity of the model was subsequently scaled back to train for a simpler dataset of 300 on the S-shaped environment to see if it was possible to get a workable proof-of-concept for the ID model. This would allow for the RNN to focus solely on learning a single path. The variance of the dataset is decreased at the loss of removing the diversity of environments, however this should allow for training to occur faster on the single case as that of the IK model proof-of-concept.

3.6 Results - Inverse Dynamics

The result for training for 8000 epochs after 14.5 hours is shown with the MSE loss and learning rate vs. epochs in Figure 18 below for the 300 sample dataset:

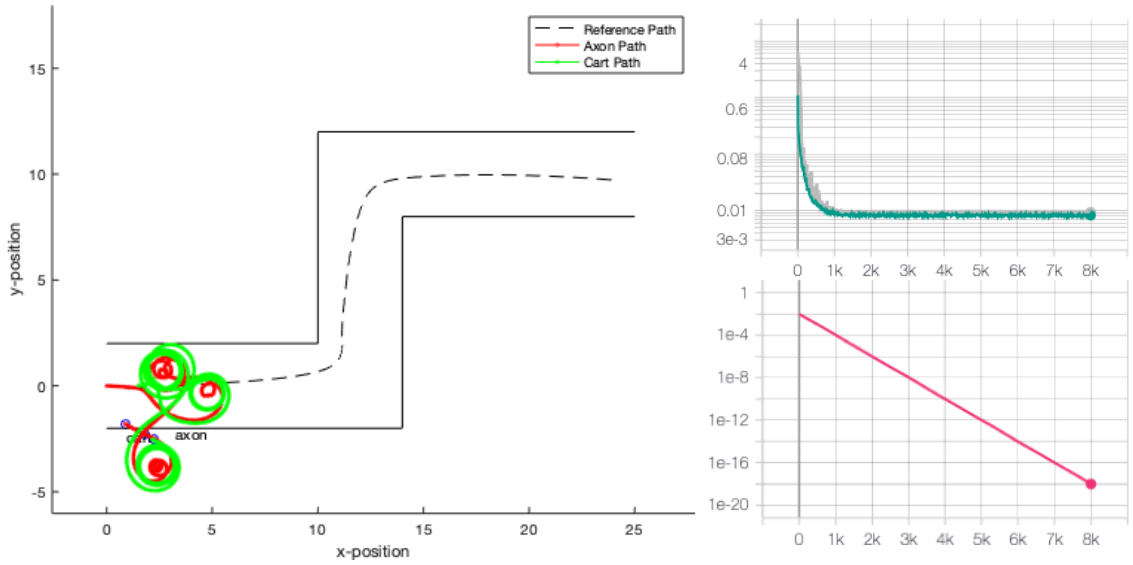


Figure 18: NN result 1 (left) with MSE loss (top)/Learning Rate (bottom) vs. epochs.

It is evident that training an RNN on the ID model proves challenging to the RNN as the complexity of the dataset is too heavy for this form of Deep Learning. Even for IK learning, the RNN struggles to minimise the error between the desired and generated linear and angular speeds. The MSE loss vs. epochs plot of Figure 18 is also representative of the average outcomes of multiple training runs. It shows that the MSE loss does not decrease with exponentially decreasing learning rate and stagnates at 0.01, with alternative training runs stagnating between two local minima (see Appendix E for example). There is no attempt to reach a global minimum and hence the training remains within the local minimum giving the result on the left. This is a symptom of stagnated training caused by the incredibly low learning rate as seen above from the steep drop in learning rate by 18 orders of magnitude. As such a second run of the training was attempted for improved learning rate scheduling on the same architecture with Figure 19 showing the results after 8000 epochs for 18.6 hours of training:

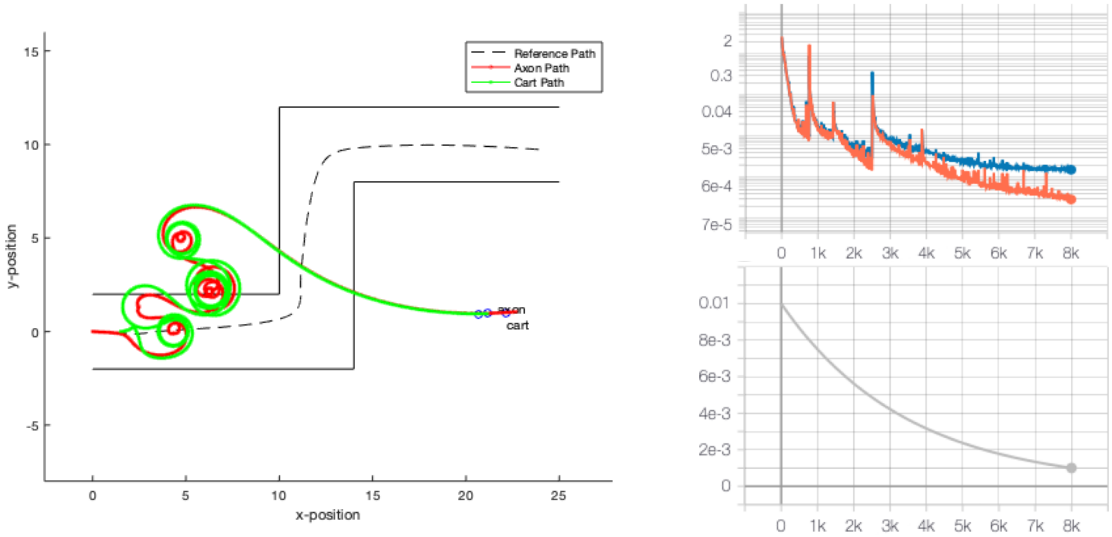


Figure 19: NN result 2 (left) with MSE loss (top)/Learning Rate (bottom) vs. epochs.

There is a clear improvement in training with a now more obvious S-shape arising out of the predicted trajectory. It shows that good conditioning of hyperparameters is necessary for adequate training. The MSE loss also follows a promising decay trend with the best value of 1.5 x 10^{-3} for training and 3 x 10^{-4} for validation. The trend also promises to continue beyond 8000 epochs which teases potential for accurate results. This can be achieved with a larger NN, however implementing more layers and neurons also increases the computational demands of training with an added 50-20-20 neurons in the hidden layers predicted to take past 72 hours of training for 15000 epochs on the HPC. Pre-loading causes resets to the Adam's momentum optimisation which is valuable to steady and fast weight updates. As such, given the time span of this project, the current training approach could not be accomplished in due time and hence further measures and implications are discussed later in this report.

Granted the above, qualitatively inspecting the difference between the first iteration (Figure 16) of the ID RNN with the latest results, a few optimistic observations were made. The robot-trailer system of the current iteration does not heavily spiral outside of the environment borders as in Figure 16 on the right and, as seen by the 'loops' performed by the robot-trailer system, the RNN recognises executed turns. Comparing this to the results of the IK model for the simple case, there are 3 points at which the system turns; the initial protrusion at the start, the first turn, and the second turn. This is recognised by the 3 'loops', with the smallest loop corresponding to the initial protrusion[3]. Similar logic is also reflected in the left image of Figure 18. Henceforth, the ID RNN has the potential to further improve and must be considered with advanced Neural Network architectures and layers such as Convolutional layers.

[3]The protrusion is due to the simulation preferring to reverse the robot such that it 'pushes' the pin-joint. This defies material bounds and causes 'knifing' of the system. It is fixable by limiting the pin-joint DoF. This was not applied in the project as it does not impact the Deep Learning problem.

4 Error Compensation

The second aspect of the project aimed at combining a Kinematic Controller (KC) with Deep Learning control for error compensation within Matlab's Simulink environment. The first aspect to explore is the I/Os that are required for the Deep Learning control. A typical control scheme using positional error was considered but later rejected as the control flow of the existing control scheme does not incorporate position within Simulink and uses speed control instead[4]. The existing control scheme for the Robot-Trailer system was subsequently analysed and is shown in Figure 20 below:

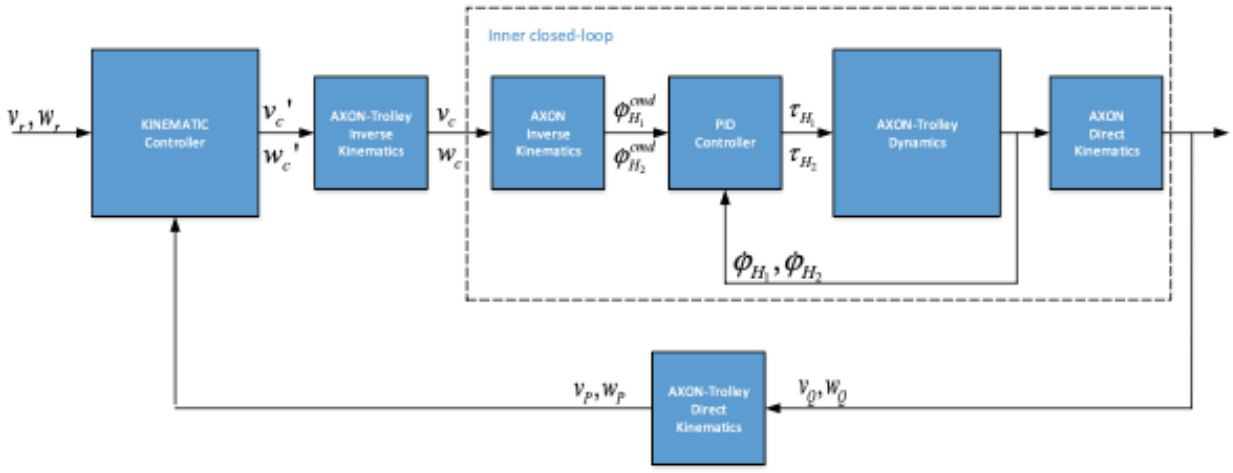


Figure 20: Control Scheme for Robot-Trailer motion [32].

From this, the KC is observed to have the inputs of the desired (reference) linear and angular speeds, V_r and W_r, and actual linear and angular speeds of the trailer, V_p and W_p, that are used to find speed errors. The outputs are subsequently the linear and angular speed commands, V_c and W_c, that are translated into the kinematics of the robot for driving the differential motors. It is hence advised that the Deep Learning Control follows a similar set of I/Os such that it is placed parallel to the Kinematic Controller. This was incorporated as shown in Figure 21:

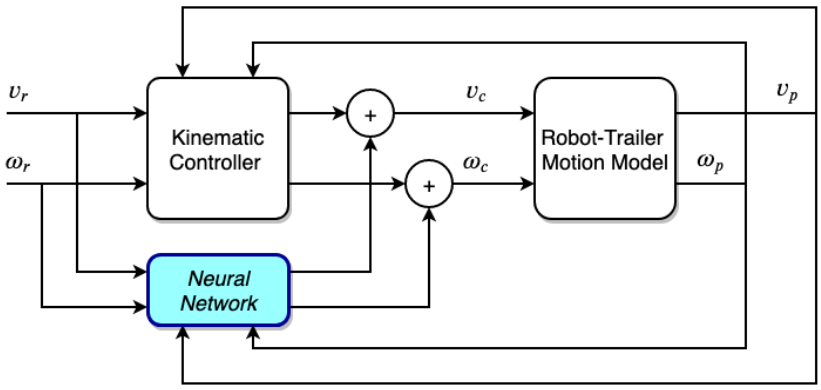


Figure 21: Neural Network Error Compensation Control Scheme.

[4]Position control was additionally incorporated as an option in the Simulink model and implemented. No improvements over speed control was observed and hence either control scheme can be selected.

The Robot-Trailer Motion Model describes the control system in Figure 15 excluding the KC. Having considered the I/Os, a multitude of approaches were explored within Python and Matlab for developing the above control scheme:

- Incorporate Python and TensorFlow live within Simulink. Problem: Porting between Matlab and Python slows down simulation if executed real-time in Simulink. Exiting a function block incorporating a Python script shuts down the Python interpreter and resets all TensorFlow training parameters.
- Pre-train an RNN in batch mode within Python to utilise specialised TensorFlow layers and migrate to Matlab. Problem: Matlab does not support specific layer types as it cannot convert the layers into an equivalent Deep Learning Toolbox model. Using simpler layers in TensorFlow and converting over is also redundant as the Deep Learning Toolbox is available.
- Use Matlab's Deep Learning Toolbox which comes with an extensive library of layer types. Problem: Simulink is built directly from C and not from Matlab. Simulink subsequently does not recognise functions such as *train* and *adapt* of the Deep Learning Toolbox. Attempts to make Deep Learning methods extrinsic also did not produce outputs from the functions. A common bypass is to use S-function of Simulink which allow code scripts in Python and Matlab to be compiled in C for use within Simulink. The limitations to this is that NVIDIA Cuda drivers are required which access NVIDIA GPUs for compilation. This limits development as the systems used are not equipped with NVIDIA GPUs and do not have updated drivers.
- Use Simulink's Neural Network models and Perceptron function blocks. Problem: Prebuilt model blocks are single input and single output models built for simple control. Perceptron function blocks are effective and provide full control over the design, however they are not scalable as the complexity of construction within Simulink increases drastically. This slows down development if the architecture is to be changed (see Appendix F for an example attempt).

The above points share the problems of software and module incompatibility and hence constrain effective use of pre-existing tools. It was subsequently decided to develop an MLP within a Matlab function block of Simulink to benefit from scalability and agile development of the control scheme. To support this, error compensation does not require specialised layers and cells as of the previous problem. This is because Deep Learning is not required to predict future outcomes, rather to naively reduce error in real-time with the KC as the simulation progresses. This contextualises the problem and defines the 2 by 3200 dataset with the reference linear and angular speeds, v_r and ω_r, as features and the time-steps, $i = 1, 2, 3, ..., n$ as samples. Each epoch is then defined as one simulation. As such the MLP was deemed to be the focus of development.

4.1 Multi-Layer Perceptron

The MLP was built to be scalable and uses Backpropagation to adjust the network's weights. The code for the MLP can be seen in Appendix G and shows the forward and backward process of training. By nature, the MLP is an adaptive regression NN as it is actively trained to reduce the error at each time-step of the simulation and must provide continuous command signals. The network is to be trained over the course of several runs until the overall MSE loss is less than that of the KC alone. It is incorporated into Simulink via Figure 22 with Appendix H showing its location in the KC:

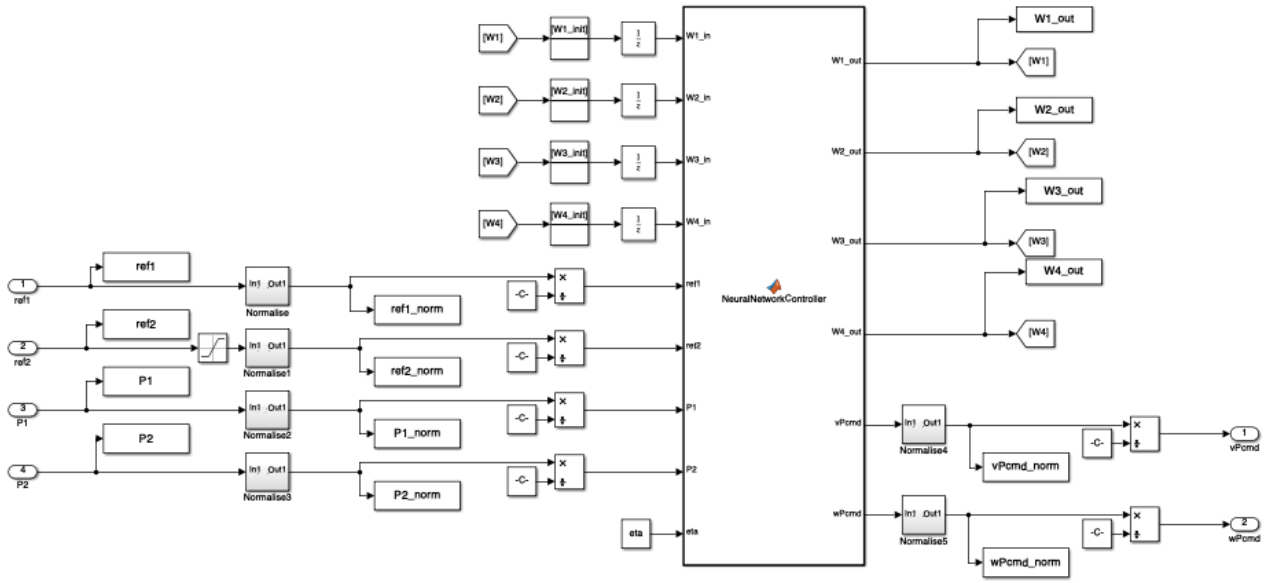


Figure 22: Simulink MLP Function Block with normalisation of inputs and outputs.

Before feeding the raw inputs into the MLP, regularisation followed by normalisation must occur. This ensures that the weights remain small as to not saturate the activations and hence allow outputs to lie within the curved regions of activation functions. The regularisation is done in accordance with Equation (10):

$$x_i' = \frac{x_i - \bar{x}}{x_{rng}\sigma} \tag{10}$$

where x_i' is the normalised input variable, x_i is the raw input variable, $\bar{x}$ is the mean of the input variable across the entire simulation, σ is the standard deviation of the input variable, and x_{rng} is the range of the input variable. This approach is repeated for post-processing the output variables of the MLP to limit sudden perturbations of the MLP regression on the speed commands. The I/O dataset from the $(n-1)^{th}$ simulation was used for the above pre-processing to find the means, standard deviations, and ranges. For the first epoch, the dataset used was from a KC only run to initialise the first iterations means, standard deviations, and ranges. Following regularisation and normalisation, the MLP function block allows initialised weights and updated weights to be passed as well as the learning rate. Architecture parameters such as number of hidden layers and neurons were also considered

as the inputs to the function blocks but not implemented as the finalised architecture need not be changed.

Looking into the function block, the MLP architecture was a 2-10-7-5-2 network with 2 inputs 3 hidden layers and 2 outputs. The inputs into the forward process of the network are the desired linear and angular speeds, v_r and ω_r (ref1 and ref2 of Figure 22), and the forward process outputs are the speed commands, v_c and ω_c (vPcmd and wPcmd of Figure 22). There are 3 hidden layers for MLP generalisation with adequate neuron count for avoiding overfitting. This gave less weights to train whilst retaining the same performance of a single layer with more neurons. The neurons use the tansig activation function which has a non-saturated range of -1 to 1 (described previously with LSTM cells). The tansig function and derivative is described by Equations (11) and (12) below and the former is shown in Figure 23 on the right:

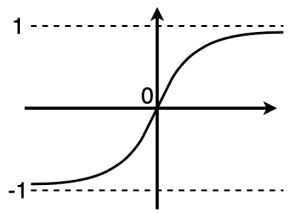


Figure 23: Tansig Activation Function.

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{11}$$

$$\frac{df(x)}{dx} = 1 - f(x)^2 \tag{12}$$

where x is the sum of the weighted inputs (or outputs of previous cells) and $f(x)$ is simply the output of that cell. This is fed forward through the network until the output neurons which have a pure linear (purelin) activation function for the regression problem. The outputs are then passed through the rest of the Simulink model to extract the actual speeds which are used in the backwards process for Backpropagation. Backpropagation subsequently uses the actual linear and angular speeds, v_p and ω_p (P1 and P2 of Figure 22), to calculate the linear and angular speed errors as in Equations (13) and (14) respectively:

$$v_e = v_r - v_p \tag{13}$$

$$\omega_e = \omega_r - \omega_p \tag{14}$$

The errors are then fed back through the network in a cascade process with error gradients calculated from higher layers. The weight update algorithm is described in Algorithm 1. Training considerations and methods were then explored. The following are incorporated for optimal training:

- Randomised weight initialisation between -0.01 and 0.01 tackles vanishing and exploding gradients within the backward pass of the training. This ensures that weights do not drastically increase or stagnate, both of which respectively cause divergence from minimum points and stopping of training towards the optimum.
- Exponential learning rate scheduling (introduced in the previous section) was applied and an initial learning rate, η_i, of 0.0005 was selected. By decreasing the learning rate, training brings the algorithm closer to the optimum without overshooting minima and gives a lower MSE loss resulting in better performance.

- Simulated Annealing was also used and proved to be effective in combating local minima entrapment. Similarly to weight initialisation, it is accomplished by adding small, randomised perturbations within the range of -0.002 and 0.002 to the weights if the difference in MSE loss between two adjacent runs is below an acceptable threshold.

Algorithm 1: Backpropagation - Multi-Layer Perceptron of control scheme.

Input: Desired and actual speeds; Outputs of all neurons in each Hidden Layer; Weights of current forward process; learning rate at current epoch;

calculate errors for linear and angular speeds;

$E = \begin{bmatrix} v_r - v_p \\ \omega_r - \omega_p \end{bmatrix}$;

calculate error gradients for each neuron of each layer;

i is Layer number, *j* is *i*th Layer's neuron number, *k* is *i-1*th Layer's neuron number;

for *each output neuron* **do**

  $\Delta E_j^4 = E_j$;

for *each Hidden Layer starting from last to first* **do**

  for *each neuron in i-1 Layer* **do**

    calculate the gradient, $\frac{dO_j^i}{dV}$, across the activation function using Equation 12;

    $\Delta E_k^i = \sum_{j=1}^{s} \frac{dO_j^i}{dV} W_{jk}^{i+1} \Delta E_j^{i+1}$;

update and save the weights between each layer for the next training sample;

for *each Output Layer neuron* **do**

  for *each neuron in i-1 Layer* **do**

    $W_{jk}^4 = W_{jk}^4 + \eta \Delta E_j^4 O_k^3$;

for *each Hidden Layer starting from last to first* **do**

  if *not first Hidden Layer* **then**

    for *each neuron in i Layer* **do**

      for *each neuron in i-1 Layer* **do**

        $W_{jk}^i = W_{jk}^i + \eta \Delta E_j^i O_k^{i-1}$;

  else

    for *each neuron in i Layer* **do**

      for *each Input neuron* **do**

        $W_{jk}^1 = W_{jk}^1 + \eta \Delta E_j^1 X_k$;

Result: Updated Weights

There were several challenges encountered during incorporation when exploring the training methods mentioned above. The range at which the weights are initialised is optimal and discovered by trial and error. Outside of this range, training is destabilised and divergence occurs with every epoch. Similarly, η_i causes instability and divergence for higher values and inversely, lower values would require much longer time frames to train and hence were

kept above 0.0001. Initially however, the learning rates of 0.000001 were discovered to be the most effective for use but regularisation and normalisation increased the value to above 0.0001, giving credence to proper data handling. The range of Simulated Annealing weight perturbations follows the same logic but were initially considered to be kept low such that any prior training is not undone by large changes to the weights.

After developing the MLP architecture and training methods in Matlab (see Appendix I for code), the positional MSE was calculated at each simulation run-through (epoch). A U-shaped path was the baseline used with a scale 3x smaller than in Inverse Model Learning. This is to bring out larger errors naturally for training the MLP. The MSE loss of each run was compared to that of the KC only simulation to observe whether the KC + MLP control perform better than KC only. Subsequently, the MSE loss of the KC simulation was found to be 171.9726 without normalising positional variables. The training procedure with trajectories are plotted in Figure 24 with robot-trailer system starting in the top-left corner and ending in the top-right corner of the plot:

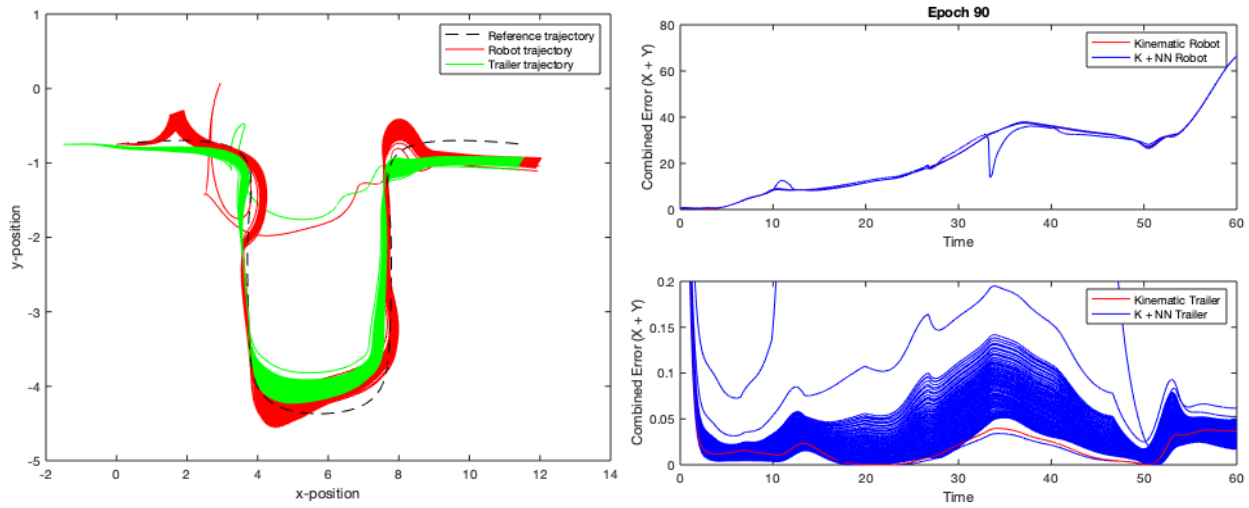


Figure 24: Training Procedure for the KC + MLP Control Scheme, path generation (left) and MSE losses (right).

The trajectory trend attempts to force the trailer closer to the reference trajectory by lessening the MSE loss of the right figure above. The blue (KC + MLP) MSE loss vs Time line is decreased from curves with higher values at all points to the red line (KC only). Note the trailer trajectory correction towards the reference path at the beginning and end of the trajectories past the U-turn. The high MSE loss of the the trailer at the beginning of the trajectories is due to the trailer initialising outside of the reference path start. This is the point at which the robot initialises, hence no MSE loss in the first 5 seconds. Note also the spike of the robot MSE loss which corresponds to the anomalous path of the trajectory plot. This occurs at the first training epoch as the weights are initialised. With a higher learning rate, this would cause instability and divergence of the system. The final result of training is acquired at the onset of divergence when the MSE loss no longer decreases.

4.2 Results - Error Compensation

After training the KC + MLP control scheme for 94 epochs on average, the MSE loss of the KC + MLP control was found to be lower than that of the KC alone for 9 epochs, with the lowest MSE loss used to generate the above trajectories being 166.1706. Figure 25 displays the reference trajectory and training result, with the robot and trailer starting in the top-left corner at (0, -0.75) and (-1.5, -0.75) respectively:

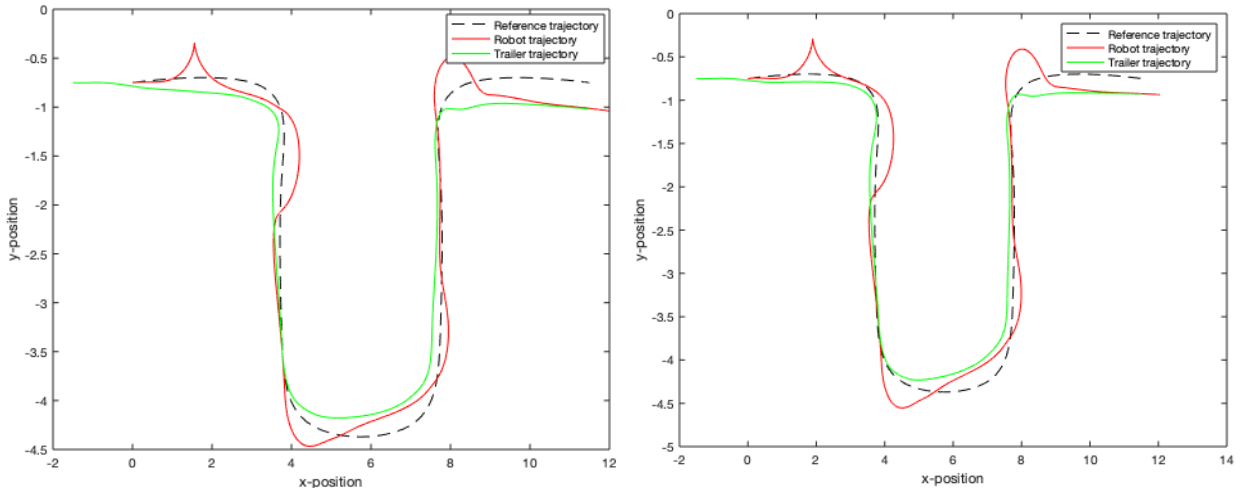


Figure 25: Trajectory x and y position plots for KC (left) and KC + MLP (right).

The above shows a minor improvement to the control scheme, with main improvements occurring at the beginning and end segments of the simulation as the trailer more closely follows the reference path. However, this improvement is not heavily evident within the middle segment of the path. This is confirmed by the training trend shown in Figure 24 discovered previously, with the start and end segments for the trailer 'moving up' and the middle 'moving down' with the concurrent MSE loss decreasing at all points. The improvement is subsequently validated by the MSE loss vs time plots of Figure 26 below for both the robot and the trailer:

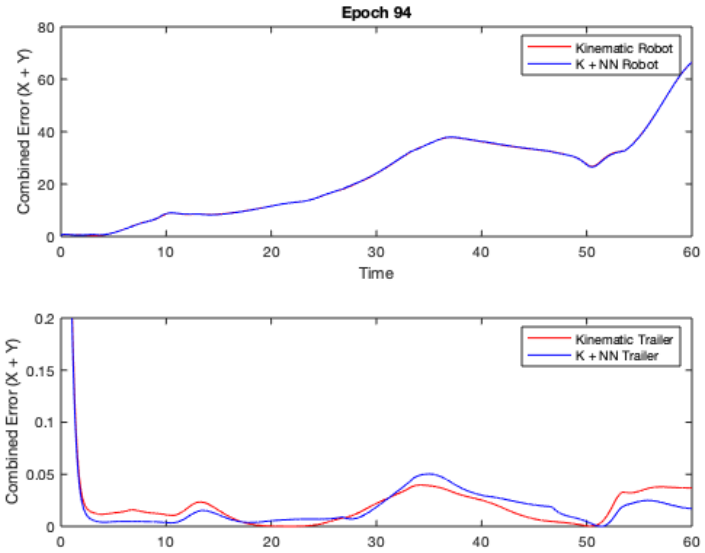


Figure 26: MSE loss of KC and KC + MLP of the robot and trailer.

As the MSE loss of the robot for the KC + MLP control is equal to that of KC control, the robot MSE loss has not changed and hence there is no improvement to the speed and trajectory following of the robot. This is expected as the speeds that are being controlled are that of the trailer and hence there should be minute change between the desired position and the actual robot position (hence the blue line exactly intersecting the red line of the MSE losses in Figure 26). The MSE loss for the trailer, however, has improved as seen at the beginning 18 seconds of the run and the final 9 seconds. These two contribute the most reduction in MSE loss of the KC + MLP control system. It is hence conclusive that the addition of the MLP improves the system performance and can hence be deployed within the robot-trailer control application for this iteration of the KC and configuration.

Looking at the middle U-shaped segment, performance deteriorates when comparing the two control schemes. This is supported by the trailer MSE loss plot in between 30-52 seconds which corresponds to the U-shaped curve. This can be due to the small value of the MSE weights which are more optimised for the start and end of the trajectory. The MLP also does not have the capacity to predict and adapt to the speed change as it has no capacity for memory like an LSTM cell hence it cannot learn the trajectory patterns to make an adequate change in the future curve. Further training was attempted to minimise the U-shape MSE error, however divergence emerges and the MLP performance degrades.

One must also consider external changes on the MLP training. As the KC + MLP control system was only trained on one path, there is no reference on how the MLP would perform within other environments. The MLP control signals run the risk of over- and under-compensating which is evidenced by the poor performance within the U-shape as the error here is comparatively large when compared to the whole path. The MLP also shows degraded performance within this region and hence must also be more heavily considered for minimising robot MSE loss. The MLP performance is hence predicted to be poor within sharp corners and hence sharper environments must be used to train the MLP to force out larger values and derivatives of error.

5 Discussion

Considering IK and ID learning, it is feasible to learn simple systems for one considered path. It shows the capabilities of RNNs to learn the inverse models, however scalability within other environments is impacted by the variance of the dataset. This can be beneficial for RNNs in recognising more ubiquitous patterns. However, the benefit only holds true for large scale datasets with sample sizes of 10000+ samples. The size of the current datasets (300 and 1000) will have some natural biases, especially with certain environments generating more false paths which are subsequently removed. This bias hence pushes the NN towards the dominant paths. The characteristics of the S- and U-shaped paths are also not well extracted, recognised, and combined by the RNN. This is seen by the complex IK model deviating from the desired path as it progresses with an MSE loss of $1.2x10^{-4}$ (which is deceptively low but still inadequate). The deviation is caused by the S- shaped path being well-learnt in the beginning half of the simulation causing good completion. Beyond the midpoint, the RNN has half the data of only the SNS-shaped path to complete the simulation, with the U-shaped path not inverted to an N-shaped path and combined with the data. In hindsight, since the RNNs do not classify the paths that are presented, there is little room to expect identification of the U-shaped path for use as an N-shaped path. The RNN only detects low-level features of when and how to turn to create a speed schedule based on the input path which is not applicable in new environments.

It has also been recognised that LSTM cells struggle at learning longer sequences of 100 time-steps. Subsequently, the Conv1D layer was added to alleviate the pressure on LSTM layers and have more meaningful interpretations of the data such as straight paths and 90 degree turns. The marginal improvement shows that patterns can be learnt, however more intelligent architectures should be used to extract higher level representations - possibly by using WaveNet [33] which can decipher frequency information of audio using Convolutional layers. This can be used to detect periodic torque patterns within the dataset and reduce the noise level of the torque readings. It must also be noted that the kernel size does not impact training to the extent expected as a wider range of inspected time-steps can lead to too much generalisation. In hindsight, Convolutional layers should have been further explored to select the correct layer parameters. Within TensorFlow, zero padding and strides should be more precisely tuned to generate shorter input sequences (less than 100 time-steps) for the LSTM layers to handle.

The RNN for the ID model is also not expected to learn more difficult environments due to the large variance of data within the dataset. The best results achieved are of the simple dataset for 1 environment with a coherent attempt at path following at an MSE loss of $3x10^{-4}$. To take what was discovered from the simple cases, the IK and ID problems can be subdivided into more discrete components in supervised control. It was recommended to subdivide the path into smaller components and use NN clusters to control the robot within

those segments. Here, a balance must be struck between simplicity and utility of networks as those which are too simple can be made redundant by conventional control schemes and those that are too complex can cause the problems and deviations encountered in this project. Perhaps ID control is also made redundant as IK control can adequately generate the speeds necessary to traverse the path for which conventional approaches or KC + MLP control can then follow.

The MLP has proven to be adequate in minimising error of the KC. The obvious flaw is the U-shaped turn of the path in which the MSE loss does not go lower than of the KC alone. This point is also that of highest error and hence the MLP under-compensates for large errors. The best solution also approaches a local minimum and hence further stochastic measures should be taken to escape the approach. This is already implemented by Simulated Annealing, however this approach is pseudo-random and cannot direct the MLP in the right hyper-plane direction. Training methods such as momentum minimisation through optimisers such as Adam or AdaGrad can subsequently supplement for speed towards such minima. The MLP can also be combined with a Lagrangian approach for constrained optimisation given the systems physical constraints to further improve performance (as introduced by Chen et al.) A constrained approach of taking environment borders into account (or maximum dimensions of the robot/trailer) can also be explored. The approach will limit the MSE loss such that divergence and updates with exploded gradients do not impact the training which is evident from the MSE loss trend of Figure 24. The constraint is to be incorporated into the Lagrangian cost function with the multipliers behaving as optimisation variables.

Taking a look at the project conduct, the iterative method of Inverse Model Learning proved to be slow in achieving results when compared to Error Compensation. The main limits to successful results was lack of consideration for more intelligent approaches and the scale of training as some runs took several days to train without improvement over previous tasks. With limited GPUs available to execute multiple runs, one can only run 2 simulations simultaneously at best. However GPU usage has clearly sped up training and must be credited for that as it would not have been possible to train the current models on i7 CPUs. Fine tuning methods such as dataset size should have been conducted to decrease the sequence length from 3200 down to 1000, however the solver methods within Simulink would give more inaccurate, lower-order results. Using legacy Matlab code to generate the dataset could have been used for simpler datasets but proved to be time-consuming as torque scheduling requires severe trial and error which is only affordable in velocity scheduling (Appendix B).

6 Conclusion

The project aimed at applying Deep Learning to practical and complex robotic systems. After introducing the topic and conducting a review of the literature, the robot-trailer task was described and the system's inputs and outputs were found from its Kinematics and Dynamics. From this, Inverse Model Learning was explored. Using the models available from legacy code and Simulink, the IK and ID model learning datasets were respectively generated. The inputs to both models is the trailer position, x_n and y_n, and orientation, θ_n, and robot orientation, θ_q. The outputs are the robot linear and angular speeds, v_q and ω_q, for the IK and the left and right wheel torques, τ_1 and τ_2, for the ID. The TensorFlow RNN was then developed, trained on the i7 CPU and V100 GPU, applied to predict paths, and results analysed for both models. The IK and ID models were observed to struggle for the complex case with 3 environments, however performed fairly adequately within the S-shaped environment. The IK MSE loss was 9.48×10^{-5}, showing decent path following, and the ID MSE loss was 3×10^{-4}, showing a good regression attempt.

Error reduction for KC control within Simulink was then explored and a KC + MLP control system was developed after attempting all possible pre-designed and manually made options within Python, Matlab, and Simulink. Learning rate scheduling, weight randomisation, normalisation and regularisation, and simulated annealing were applied to optimise the performance of the MLP. The results were then presented and analysed. The KC + MLP control produced an MSE loss of 166.1706 for the best case which is less than the KC only MSE loss of 171.9726, showing improved trajectory following. The results were then discussed and rationalised whilst inspecting the project conduct. Future work is presented in the following section for the continuation of Deep Learning within robot-trailer control.

Taking a step back to look at the overall project, a few valuable insights were made into applying Deep Learning within robot control, especially for complex dynamic systems. These include the limitations of LSTM cells and the applications of Convolutional layers. As such, Deep Learning within the field of robotics is promising and must be pursued further. With applications into machine vision already deployed, Neural Networks are heavily leaking into path planning, navigation, and control as evidenced by the literature. With further development and merging of other Machine Learning methods such as Reinforcement Learning, it is only a matter of time before commercial robotics will be deployed on a large scale. Today's world will subsequently be brought closer to what was considered as science-fiction and, with due time and diligence, will help solve many of the world's most challenging issues.

(All code is available on github.com here)

7 Future Work

For Inverse Model Learning:

- Improve current Convolutional Layer implementation to shorten the vector input in time to less than 100 elements for LSTM processing. RNN cells are limited by forgetfulness beyond 100 time-steps which can be supplemented by Convolutional layers.
- Explore further implementations of Convolutional layers or LSTM layers through architectures such as WaveNet. Torque data includes noise and complex long-term patterns hence dissection of data into patterns creates value for the network in pattern combination at higher levels.
- Use leaner alternatives to the LSTM cell such as Gated Recurrent Unit (GRU) cells [34]. GRUs have leaner architectures to LSTM cells by combining the short- and long-term memory states into one vector, removing the need for multiple gates.
- Combine Deep and Reinforcement Learning through methods such as Deep Q-Learning. Using a state-agent-action-reward approach can aid in discriminating environment borders or penalise the robot for straying off-course from selected waypoints. A reward (quality value) can be given based on path fitness and an NN can be used to fit the function that maximises the reward.

For Error Compensation:

- Increase dataset variance by applying to different paths - specifically with sharp corners and turns. This trains the MLP on higher errors created when executing a turn.
- Incorporate memory based cell types for short-term speed monitoring. The MLP can then consider acceleration and deceleration of the robot-trailer and increase/decrease compensation commands proportionally.

IK and ID models should be combined with MLP control to exhaust the benefits of the MLP feedback control loop and the memory based learning of RNNs.

- Feedback control with IK/ID path generation giving an error at specific waypoints.
- Cascade control of primary IK/ID control and secondary MLP control for adaptive path updating based on deviation error.
- Master-Slave control with the RNN as the master and MLP control as the slave.

Access to a physical robot-trailer system is also highly recommended. Collaboration with the industry provider from which this project is derived should be undertaken to develop an applicable system or realistic model. Specifically, collection of raw torque data from physical tests is encouraged to model uncertain dynamics and disturbances within Simulink such that the system can be trained and ported to physical robot control.

References

[1] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Mehrali Hemmatinezhad, Mohammad Gholizadeh, Mohammadrahim Ramezaniyan, Shahram Shafiee, and Amin Ghazi Zahedi. Predicting the success of nations in asian games using neural network. 03 2020.

[3] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[4] Warren S Mcculloch and Walter Pitts. A logical calculus nervous activity. *Bulletin of Mathematical Biology*, 52(l):99–115, 1990.

[5] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[6] Simon Haykin. *Neural Networks and Learning Machines*, volume 3. 2008.

[7] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Feynman and Computation*, 79(April):7–19, 1982.

[8] S. Lawrence, C. L. Giles, Ah Chung Tsoi, and A. D. Back. Face recognition: a convolutional neural-network approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, 1997.

[9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning Internal Representations Error Propagation. *Cognitive Science*, 1(V):318–362, 1986.

[10] Andreas Griewank. Who Invented the Reverse Mode of Differentiation? *Documenta Mathematica · Extra Volume ISMP*, I:389–400, 2012.

[11] David J. C. MacKay. A Practical Bayesian Framework. *Neural Computation*, 472(1):448–472, 1992.

[12] Sebastian Ruder. An overview of gradient descent optimization algorithms. pages 1–14, 2016.

[13] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[16] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019.

[17] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. pages 1–17, 2019.

[18] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. 2019.

[19] Cecilia Di Nardi. An automatic speech recognition android app for als patients. 09 2019.

[20] Lei Tai, Jingwei Zhang, Ming Liu, Joschka Boedecker, and Wolfram Burgard. A survey of deep network solutions for learning control in robotics: From reinforcement to imitation. 2016.

[21] Jahanzaib Shabbir and Tarique Anwer. A Survey of Deep Learning Techniques for Mobile Robot Applications. 14(8):1–10, 2018.

[22] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[23] S. A. Gautam and N. Verma. Path planning for unmanned aerial vehicle based on genetic algorithm artificial neural network in 3d. In *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, pages 1–5, 2014.

[24] John J Craig. *Introduction to robotics: mechanics and control, 3/E*. Pearson Education India, 2009.

[25] Ahmed R.J. Almusawi, L. Canan Dülger, and Sadettin Kapucu. A New Artificial Neural Network Approach in Solving Inverse Kinematics of Robotic Arm (Denso VP6242). *Computational Intelligence and Neuroscience*, 2016, 2016.

[26] Athanasios S. Polydoros, Lazaros Nalpantidis, and Volker Kruger. Real-time deep learning of robotic manipulator inverse dynamics. *IEEE International Conference on Intelligent Robots and Systems*, 2015-December:3442–3448, 2015.

[27] Dechao Chen, Shuai Li, and Liefa Liao. A recurrent neural network applied to optimal motion control of mobile robots with physical constraints. *Applied Soft Computing Journal*, 85:105880, 2019.

[28] H. Wang, J. Duan, M. Wang, J. Zhao, and Z. Dong. Research on robot path planning based on fuzzy neural network algorithm. In *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pages 1800–1803, 2018.

[29] Kendrick Amezquita-Semprun, Jr Manuel Del Rosario, and Peter C.Y. Chen. Dynamics model of a differential drive mobile robot towing an off-axle trailer. *International Journal of Mechanical Engineering and Robotics Research*, 7(6):583–589, 2018.

[30] Aurelien Geron. *Hands-On Machine Learning with Scikit Learn, Keras and TensorFlow*. O'Reilly, 2019.

[31] NVIDIA. NVIDIA Tesla V100 Tensor Core GPU, 2020.

[32] Microsystem Engineering, Robotics Group, and Peter C. Y. Chen. Robotic Mobile Platform Kinematic Control. pages 1–11.

[33] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alexander Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *Arxiv*, 2016.

[34] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

Appendices

Appendix A. System parameters - roboparam.m

```
%% CONFIG5
ROBOPARAM.M COURTESY OF KHOO YI CHUN — NUS FINAL YEAR MECHANICAL ENGINEERING
    STUDENT

% Kinematic Controller Parameters
k1 = 3.0;
k2 = 3.5;
k3 = 5.5;

%AXON—Trolley Initial conditions
yref_trolley0 = 0;

%Trolley initial position
xtrolley0 = —1.5;
ytrolley0 = —0.75;
th1 = 0*pi/180;
th0 = 0*pi/180;

th0_ref = 0;
th1_ref = 0;

psicw10 = 0;
psicw20 = 0;
psicw30 = 0;
psicw40 = 0;
phicw10 = 0;
phicw20 = 0;
phicw30 = 0;
phicw40 = 0;
v0 = 0;
w0 = 0;

rho0 = 0.5;
rho1 = 1;  % distance in m

```

```
xaxon0 = xtrolley0+rho1*cos(th1)+rho0*cos(th0);
yaxon0 = ytrolley0+rho1*sin(th1)+rho0*sin(th0);
L = 0.387/2;
s_l = 0.928;
s_w = 0.48;
dcw = 0.075;
dqpx = 0;           % distance bet q and P
dqcmBx = 0*0.648/4; % distance bet Q and Axon com
cw1x = -s_l/2;
cw1_x = cw1x;
cw3x =  s_l/2;
cw3_x = cw3x;
dpcmPx = 0;%s_l/2;

% mass in kg
mB = 50;
m_P = 150;
mH = 2;
mW = 0.7;
mCW = 5.0;
mT = m_P + 2*mW + 2*mH + 2*mCW +mB;

% wheel radius
rhoH = 0.075;
rho_h = rhoH;
rhow = 0.06;
rho_w = rhow;
rhocw = 0.05;
rho_cw = rhocw;

e1 = (2*mH + mB)*rho1 + dpcmPx*m_P - 2*mCW*s_l;
e2 = (2*mH + mB)*rho0 + dqcmBx*mB;

IWy = (1/2)*mW*rho_w^2;
IWz = (1/12)*mW*(3*rho_w^2 + 0.03^2);
ICWz = (1/12)*mCW*(3*rho_cw^2 + 0.01^2);
IPz = (1/12)*m_P*(s_l^2 + (2*s_w)^2);
IBz = (1/12)*mB*(0.3^2 + 0.6^2);
IHz = (1/12)*mH*(3*rho_h^2 + 0.03^2);
```

```
IHy = (1/2)*mH*rho_h^2;

ITw = IWz + s_w^2*mW;
ITCWz = ICWz + mCW*dcw^2;
ITB = IBz + mB*dqcmBx^2;
ITH = IHz + mH*L^2;
IT_P = IPz + m_P*dpcmPx^2;

ITz = (2*mH + mB)*rho0^2 + 2*dqcmBx*mB*rho0 + ITB + 2*ITH;
ITP = (2*mH + mB)*rho1^2 + IT_P + 2*ITw + 2*mCW*(s_w^2 + s_l^2);
```

Appendix B. Dataset Generation Code

A. Kinematics

```
%% Simulation script for TTMR
% Prerequisites:
%   1. Configure the robot parameters component (mass, position, and
%   dimensions) at 'roboparam.m'
% Procedure:
%   1. Adjust dataset_size and execute
% Outputs
%   v, w -> Speeds
%   x, y, th1, th0 -> States

%% Simulation
close all;
clear all;
clc;

%% Simulation variables, parameters, and options
activate_environment = 0;
dataset_size = 300;
MSE_lim = 150; % path rejection criteria for dataset generation

roboparam;  % Lists the robot parameters necessary for the simulation

% simulation time
t0 = 0;         % initial time
tint = 0.025;   % time increments
```

```
26 tf = 80;        % final time
27
28 t = t0:tint:tf;
29
30 %% Dataset Generation
31 global set_path; % This is passed over to Simulink to generate the path with
       its pre- and post-function script
32 MSE = [];
33 for i = 1:dataset_size
34     if i == 1
35         set_path = 1;
36     elseif i == dataset_size
37         set_path = 3;
38     else
39         randomiser = 1+int16(2*rand);
40         set_path = randomiser;
41     end
42     if activate_environment == 1
43         gen_environment(set_path);
44     end
45     sim('config5sim_KinematicController_generator.slx');
46     %% INITIAL SIMULINK MODEL COURTESY OF KHOO YI CHUN - NUS FINAL YEAR
          MECHANICAL ENGINEERING STUDENT
47     MSE_loss = sum((1/2)*((x_ref-xP).^2 + (y_ref-yP).^2));
48     MSE = cat(2, MSE, MSE_loss);
49     figure(4)
50     plot([0:i-1], MSE, '-or');
51     axis([0 inf 0 inf])
52     if i == 1
53         if MSE_loss > MSE_lim
54             continue
55         end
56         xout = xP;
57         yout = yP;
58         th1out = th(:,1);
59         th0out = th(:,2);
60         Vout = vQ;
61         Wout = wQ;
62     else
```

```
        if MSE_loss > MSE_lim
            continue
        end
        xout = cat(2, xout, xP);
        yout = cat(2, yout, yP);
        th1out = cat(2, th1out, th(:,1));
        th0out = cat(2, th0out, th(:,2));
        Vout = cat(2, Vout, vQ);
        Wout = cat(2, Wout, wQ);
    end
end

% Variable Saving
csvwrite('Datasets_Kinematics/x.csv', xout);
csvwrite('Datasets_Kinematics/y.csv', yout);
csvwrite('Datasets_Kinematics/th1.csv', th1out);
csvwrite('Datasets_Kinematics/th0.csv', th0out);
Vw = [Vout, Wout];
csvwrite('Datasets_Kinematics/Vw.csv', Vw);
```

B. Dynamics

```
%% Simulation script for TTMR
% Prerequisites:
%   1. Configure the robot parameters component (mass, position, and
%   dimensions) at 'roboparam.m'
% Procedure:
%   1. Adjust dataset_size and execute
% Outputs:
%   tau1, tau2 -> Torques
%   x, y, th1, th0 -> States

%% Simulation
close all;
clear all;
clc;

%% Simulation variables, parameters, and options
activate_environment = 0;
dataset_size = 1000;
```

```
MSE_lim = 150; % path rejection criteria for dataset generation

roboparam;  % Lists the robot parameters necessary for the simulation

t0 = 0;             % initial time
tint = 0.025;       % time increments
tf = 80;            % final time
t = t0:tint:tf;     % time vector

%% Dataset Generation
global set_path;
MSE = [];
for i = 1:dataset_size
    if i == 1
        set_path = 1;
    elseif i == dataset_size
        set_path = 3;
    else
        randomiser = 1+int16(2*rand);
        set_path = randomiser;
    end
    if activate_environment == 1
        gen_environment(set_path);
    end
    sim('config5sim_KinematicController_generator.slx');
    %% INITIAL SIMULINK MODEL COURTESY OF KHOO YI CHUN - NUS FINAL YEAR
       MECHANICAL ENGINEERING STUDENT
    MSE_loss = sum((1/2)*((x_ref-xP).^2 + (y_ref-yP).^2));
    MSE = cat(2, MSE, MSE_loss);
    figure(4)
    plot([0:i-1], MSE, '-or');
    axis([0 inf 0 inf])
    if i == 1
        if MSE_loss > MSE_lim
            continue
        end
        xout = xP;
        yout = yP;
        th1out = th(:,1);
```

```
        th0out = th(:,2);
        tau1out = tau1;
        tau2out = tau2;
    else
        if MSE_loss > MSE_lim
            continue
        end
        xout = cat(2, xout, xP);
        yout = cat(2, yout, yP);
        th1out = cat(2, th1out, th(:,1));
        th0out = cat(2, th0out, th(:,2));
        tau1out = cat(2, tau1out, tau1);
        tau2out = cat(2, tau2out, tau2);
    end
end

% Variable Saving
csvwrite('Datasets_Dynamics/1env/x.csv', xout);
csvwrite('Datasets_Dynamics/1env/y.csv', yout);
csvwrite('Datasets_Dynamics/1env/th1.csv', th1out);
csvwrite('Datasets_Dynamics/1env/th0.csv', th0out);
tau = [tau1out, tau2out];
csvwrite('Datasets_Dynamics/1env/tau.csv', tau);
```

Appendix C. Velocity Scheduling - ControlV.m

```
%% Use this to input Speed and Heading of the robot
function [v, w] = controlV(t)
    a = 0.015;
    b = 0.03;
    c = 0.0035;
    d = 0.007;
    v(t>=0) = 0.4 - a + b*rand_seed(t>=0); % SPEED
    w(t<=20) = 0 - c + d*rand_seed(t<=20); % HEADING
    w(t>20 & t<=28) = 0.19 - c + d*rand_seed(t>20 & t<=28);
    w(t>28 & t<=43) = 0 - c + d*rand_seed(t>28 & t<=43);
    w(t>43 & t<=51) = -0.1925 - c + d*rand_seed(t>43 & t<=51);
    w(t>51) = 0 - c + d*rand_seed(t>51);
end
```

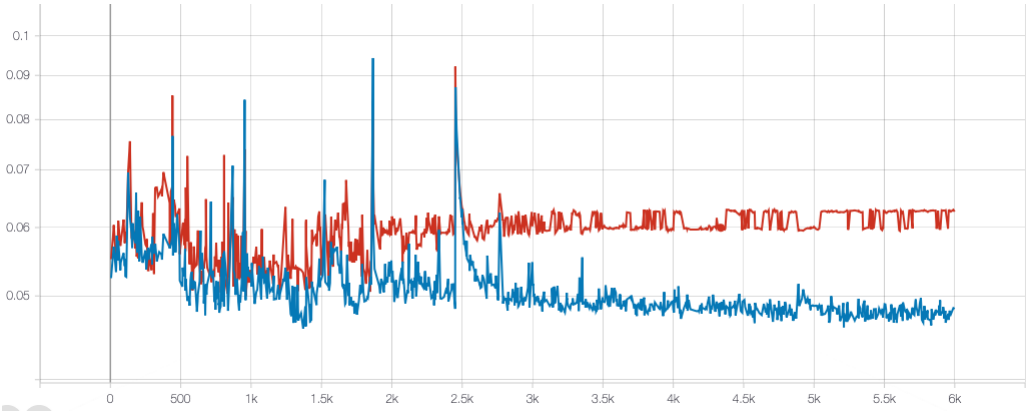
Appendix D. ID/IK Neural Network Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import os
from tensorflow import keras
import tensorflow as tf

epochs = 8000

## Dataset preparation and segmentation
# X for Dynamics
x_pos = np.array(pd.read_csv('Datasets_Dynamics/x.csv'))
y_pos = np.array(pd.read_csv('Datasets_Dynamics/y.csv'))
th1_head = np.array(pd.read_csv('Datasets_Dynamics/th1.csv'))
th0_head = np.array(pd.read_csv('Datasets_Dynamics/th0.csv'))
X = np.array([x_pos, y_pos, th1_head, th0_head]).T
print(X, X.shape)

# # X for Kinematics
# x_pos = np.array(pd.read_csv('Datasets_Kinematics/x.csv'))
# y_pos = np.array(pd.read_csv('Datasets_Kinematics/y.csv'))
# th1_head = np.array(pd.read_csv('Datasets_Kinematics/th1.csv'))
# th0_head = np.array(pd.read_csv('Datasets_Kinematics/th0.csv'))
# X = np.array([x_pos, y_pos, th1_head, th0_head]).T
# print(X, X.shape)

# Y for Dynamics
tau1_tau2 = np.array(pd.read_csv('Datasets_Dynamics/tau.csv'))
length = tau1_tau2.shape[1]
slice = int(length/2)
tau1 = tau1_tau2[:, :slice]
tau2 = tau1_tau2[:, slice:]
print(tau1.shape, tau2.shape)
Y = np.array([tau1, tau2]).T
print(Y, Y.shape)

# # Y for Kinematics
# speed_heading_speed = np.array(pd.read_csv('Datasets_Kinematics/Vw.csv'))
# length = speed_heading_speed.shape[1]
# print(speed_heading_speed, length)
# slice = int(length/2)
# speed = speed_heading_speed[:, :slice]
# heading_speed = speed_heading_speed[:, slice:]
```

```
# Y = np.array([speed, heading_speed]).T
# print(Y, Y.shape)

# # segmentation
print(batch_size)
train_size = int(batch_size * 0.7)
valid_size = int(batch_size * 0.9)
last_one = int(batch_size - 1)

print(train_size, valid_size)
X_train, Y_train = X[:train_size,:], Y[:train_size,:]
print(X_train.shape, Y_train.shape)
X_valid, Y_valid = X[train_size:valid_size,:], Y[train_size:valid_size,:]
print(X_valid.shape, Y_valid.shape)
X_test, Y_test = X[valid_size:,:], Y[valid_size:,:]
print(X_test.shape, Y_test.shape)

# # Logger
root_logdir = os.path.join(os.curdir, 'Dynamics_logs') # 'Kinematics_logs'
def get_run_logdir():
    import time
    run_id = time.strftime('run_%Y_%m_%d-%H_%M_%S')
    return os.path.join(root_logdir, run_id)
run_logdir = get_run_logdir()
file_writer = tf.summary.create_file_writer(run_logdir + '/metrics')
file_writer.set_as_default()

# # Learning Rate Scheduler
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        lr = lr0 * 0.1 ** (epoch/s)
        tf.summary.scalar('learning rate', data=lr, step=epoch)
        return lr
    return exponential_decay_fn
exponential_decay_fn = exponential_decay(lr0=0.01, s=epochs)

# # Callbacks
checkpoint_cb = keras.callbacks.ModelCheckpoint('Models/ControlRNN_Dynamics.
                                        h5', save_freq=20)
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
callbacks = [tensorboard_cb, checkpoint_cb, lr_scheduler]

# # Deep Recurring Neural Network

# Neuron dimension heads-up
batch_size = X_train.shape[0]
```

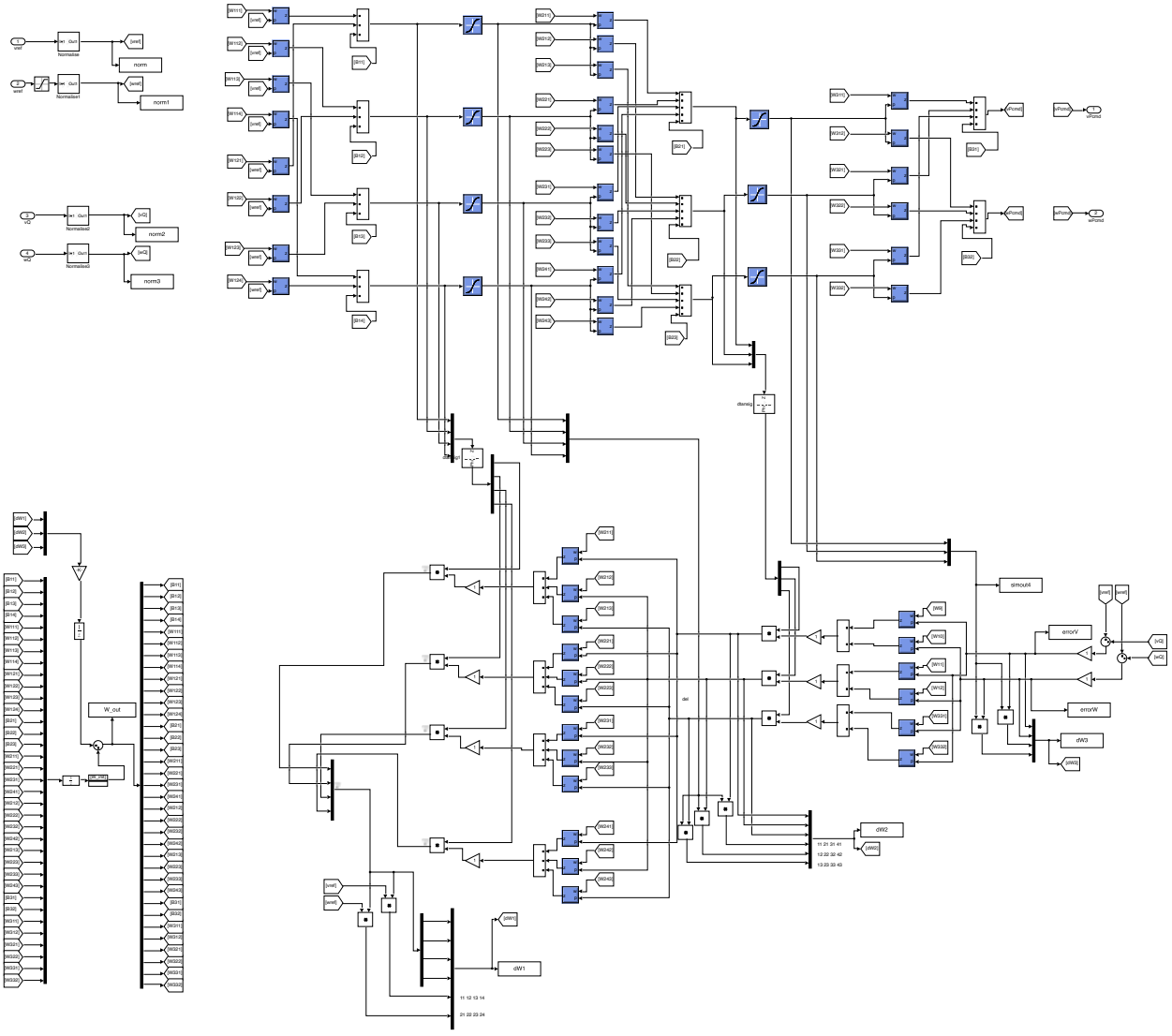
```
dim_in = X_train.shape[2]
dim_out = Y_train.shape[2]
nb_units = 10

# Use model.add(keras.layers.Conv1D(filters=20, kernel_size=20, strides=1,
                                    padding='same', input_shape=(None,
                                    dim_in)))
# for Convolutional Input Layer
model = keras.models.Sequential()
model.add(keras.layers.Bidirectional(keras.layers.LSTM(batch_input_shape=(
                                    batch_size, None, dim_in),
                                    return_sequences=True, units=nb_units)
                                    ))
model.add(keras.layers.Bidirectional(keras.layers.LSTM(50, return_sequences=
                                    True)))
model.add(keras.layers.Bidirectional(keras.layers.LSTM(30, return_sequences=
                                    True)))
model.add(keras.layers.Bidirectional(keras.layers.LSTM(10, return_sequences=
                                    True)))
model.add(keras.layers.TimeDistributed(keras.layers.Dense(dim_out)))
model.compile(loss='mean_squared_error', optimizer=optimiser)
history = model.fit(X_train, Y_train, epochs=epochs, validation_data=(
                                    X_valid, Y_valid), callbacks=callbacks
                                    )
mse_test = model.evaluate(X_test, Y_test)
Y_pred = model.predict(X[-1:,:])

print(Y_pred, Y_pred.shape)
print(mse_test)

np.savetxt('Datasets_Kinematics/PredictedSpeeds.csv', np.squeeze(Y_pred,
                                    axis=0), delimiter=',')
np.savetxt('Datasets_Dynamics/PredictedTorques.csv', np.squeeze(Y_pred, axis
                                    =0), delimiter=',')

# For use on Personal Computers (HPC does not do anything with this code)
# def plotting(history):
#     plt.plot(history.history['loss'], color = 'red')
#     plt.plot(history.history['val_loss'], color = 'blue')
#     red_patch = mpatches.Patch(color='red', label='Training')
#     blue_patch = mpatches.Patch(color='blue', label='Test')
#     plt.legend(handles=[red_patch, blue_patch])
#     plt.xlabel('Epochs')
#     plt.ylabel('MSE loss')
#     plt.show()
# plotting(history)
```

Appendix E. Stagnated MSE loss runs



Appendix F. Simulink MLP with Backpropagation

Note: Zoom in to inspect details



Appendix G. Matlab MLP with Backpropagation Function

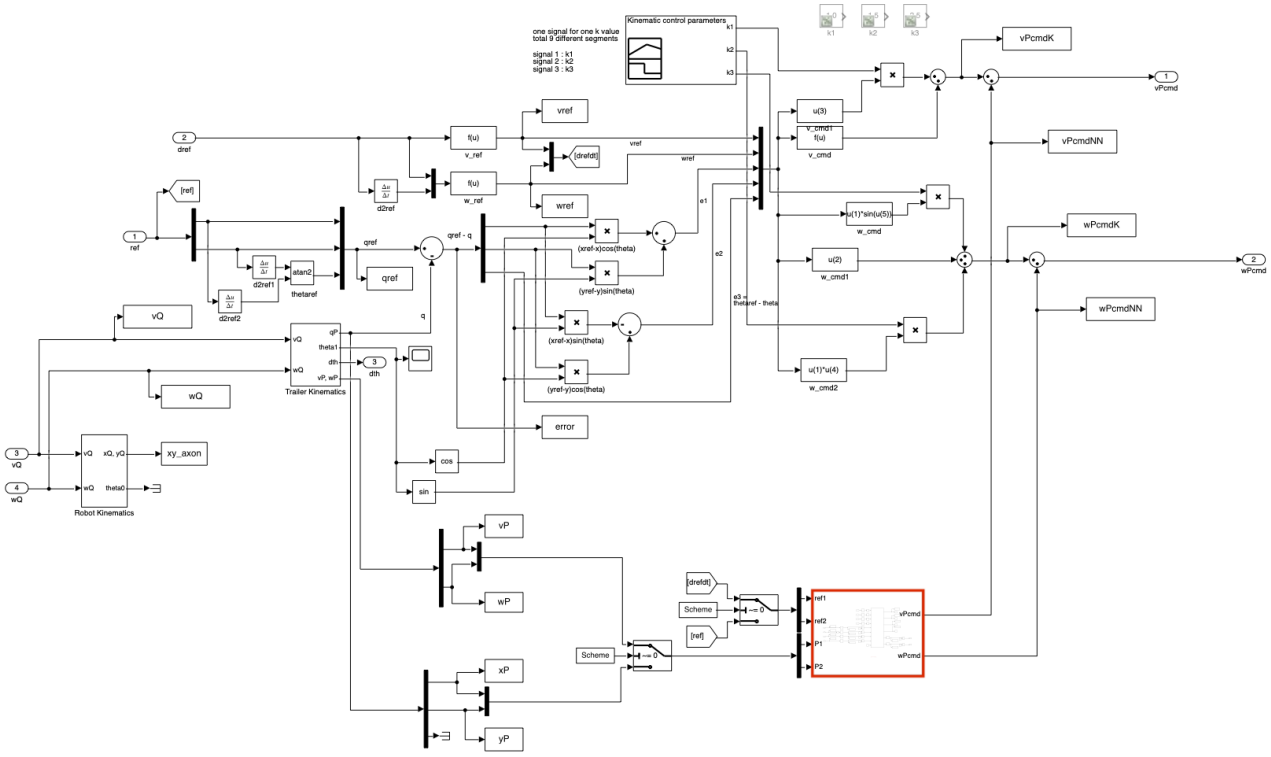
```
%% MLP Function and Backpropagation
function [W1_out,W2_out,W3_out,W4_out,vPcmd,wPcmd] = NeuralNetworkController(
    W1_in,W2_in,W3_in,W4_in,ref1,ref2,P1,P2,eta)
    % Network parameters
    In = 2;
    H1 = 10; %20;
    H2 = 7; %15;
    H3 = 5; %13;
    Out = 2;

    % Initialise Weights
    W1 = W1_in;
    W2 = W2_in;
    W3 = W3_in;
    W4 = W4_in;

    % Forward Process
    X = [1; ref1; ref2];
    E = [ref1 - P1; ref2 - P2];

    % Hidden 1
    V1 = zeros(1, H1);
    O1 = zeros(1, H1);
    for i = 1:H1
        V1(i) = W1(i,:)*X;
        O1(i) = 2/(1+exp(-2*V1(i)))-1;
    end

    % Hidden 2
    V2 = zeros(1, H2);
    O2 = zeros(1, H2);
    for i = 1:H2
        V2(i) = W2(i, :)*[1; O1'];
        O2(i) = 2/(1+exp(-2*V2(i)))-1; % tansig function for Simulink
    end

    % Hidden 3
```

```
    V3 = zeros(1, H3);
    O3 = zeros(1, H3);
    for i = 1:H3
        V3(i) = W3(i, :)*[1; O2'];
        O3(i) = 2/(1+exp(-2*V3(i)))-1; % tansig function for Simulink
    end

    % Output
    V4 = zeros(1, Out);
    Y = zeros(1, Out);
    for i = 1:Out
        V4(i) = W4(i, :)*[1;O3'];
        Y(i) = V4(i);
    end

    % Send out Velocity Commands
    vPcmd = Y(1);
    wPcmd = Y(2);

    % Backward Process
    De4 = zeros(1, Out); %Initialise for Simulink
    for i=1:Out
        De4(i) = E(i);
    end
    De3 = zeros(1, H3);
    for i=1:H3
        De3(i) = (1-(2/(1+exp(-2*O3(i)))-1)^2)*(De4*W4(:,i+1));
        %tansig derivative for Simulink
    end
    De2 = zeros(1, H2);
    for i=1:H2
        De2(i) = (1-(2/(1+exp(-2*O2(i)))-1)^2)*(De3*W3(:,i+1));
        %tansig derivative for Simulink
    end
    De1 = zeros(1, H1);
    for i=1:H1
        De1(i) = (1-(2/(1+exp(-2*O1(i)))-1)^2)*(De2*W2(:,i+1));
    end

```

```
    % Update
    for i=1:Out
        W4(i,:) = W4(i,:) + eta*De4(i)*[1;O3'];
    end
    for i=1:H3
        W3(i,:) = W3(i,:) + eta*De3(i)*[1;O2'];
    end
    for i=1:H2
        W2(i,:) = W2(i,:) + eta*De2(i)*[1;O1'];
    end
    for i=1:H1
        W1(i,:) = W1(i,:) + eta*De1(i)*X(:)';
    end

    %Post Process
    W4_out = W4;
    W3_out = W3;
    W2_out = W2;
    W1_out = W1;
```

Appendix H. MLP Function block location in KC

Note: Zoom to enhance detail



Appendix I. Training Code - Main.m

```
%% Main Training Code
clc;
clear all;
close all;

%% Code and Network Parameters
% 0 = Position Control
% 1 = Velocity Control
Scheme = 1;
In = 2;
H1 = 10
H2 = 7
H3 = 5
Out = 2;
coeff_rand = 0.01;
coeff_arand = 0.02; % 0.03
eta0 = 0.0005;  % 0.0001 for 1 0.0000001 for 0
s = 20;
epochs = 200;
t_steps = 6001;

sim('config5sim_KinematicController.slx');
disp('Start')

%% Prework
if Scheme == 0
    ref1 = x_ref;
    ref2 = y_ref;
    P1 = xP;
    P2 = yP;
elseif Scheme == 1
    ref1 = vref;
    ref2 = wref;
    P1 = vP;
    P2 = wP;
else
    error('Scheme Not Found')
```

```
end

    % Mean and Standard Deviation
ref1_mean = mean(ref1);
ref2_mean = mean(ref2);
P1_mean = mean(P1);
P2_mean = mean(P2);

ref1_std = sqrt(var(ref1));
ref2_std = sqrt(var(ref2));
P1_std = sqrt(var(P1));
P2_std = sqrt(var(P2));

vPcmd_mean = mean(vPcmd);
wPcmd_mean = mean(wPcmd);
vPcmd_std = sqrt(var(vPcmd));
wPcmd_std = sqrt(var(wPcmd));

    % Normalising via centre about mean
ref1_norm = (ref1(:) - ref1_mean)/ref1_std;
ref2_norm = (ref2(:) - ref1_mean)/ref2_std;
P1_norm = (P1(:) - P1_mean)/P1_std;
P2_norm = (P2(:) - P2_mean)/P2_std;
vPcmd_norm = (vPcmd(:) - vPcmd_mean)/vPcmd_std;
wPcmd_norm = (wPcmd(:) - wPcmd_mean)/wPcmd_std;

    % Minimum, Maximum, and Range for Scaling
ref1_min = min(ref1_norm);
ref1_max = max(ref1_norm);
ref2_min = min(ref2_norm);
ref2_max = max(ref2_norm);
P1_min = min(P1_norm);
P1_max = max(P1_norm);
P2_min = min(P2_norm);
P2_max = max(P2_norm);
vPcmd_max = max(vPcmd_norm);
vPcmd_min = min(vPcmd_norm);
wPcmd_max = max(wPcmd_norm);
wPcmd_min = min(wPcmd_norm);
```

```
77
78  ref1_rng = ref1_max - ref1_min;
79  ref2_rng = ref2_max - ref2_min;
80  P1_rng = (P1_max - P1_min);
81  P2_rng = (P2_max - P2_min);
82  vPcmd_rng = (vPcmd_max - vPcmd_min);
83  wPcmd_rng = (wPcmd_max - wPcmd_min);
84
85  %% Trailer MSE
86      % Position
87  xQMSElossK = sum((x_ref - xQ).^2)/t_steps;
88  yQMSElossK = sum((y_ref - yQ).^2)/t_steps;
89  QMSElossK_pos = xQMSElossK + yQMSElossK;
90      % Velocity
91  vQMSElossK = sum((vref - vQ).^2)/t_steps;
92  wQMSElossK = sum((wref - wQ).^2)/t_steps;
93  QMSElossK_vel = vQMSElossK + wQMSElossK;
94
95  %% Robot MSE
96      % Positions
97  xPMSElossK = sum((x_ref - xP).^2)/t_steps;
98  yPMSElossK = sum((y_ref - yP).^2)/t_steps;
99  PMSElossK_pos = xPMSElossK + yPMSElossK;
100     % Velocities
101 vPMSElossK = sum((vref - vP).^2)/t_steps;
102 wPMSElossK = sum((wref - wP).^2)/t_steps;
103 PMSElossK_vel = vPMSElossK + wPMSElossK;
104
105 %% Simulation time MSE
106 XYQMSEK = (1/2)*((x_ref - xQ).^2 + (y_ref -yQ).^2);
107 XYPMSEK = (1/2)*((x_ref - xP).^2 + (y_ref -yP).^2);
108
109 % Initialize
110 PMSEKNN_pos = [];
111 PMSEK_pos = [];
112 QMSEK_pos = [];
113 QMSEKNN_pos = [];
114
115 PMSEKNN_vel = [];
```

```
PMSEK_vel = [];
QMSEK_vel = [];
QMSEKNN_vel = [];
eta_new = eta0;
sumse = [];
for i = 1:epochs
    eta = eta_new;
    eta_new = eta0*0.1^(i/s);
    if i == 1
        W1_i = coeff_rand*(rand(H1,In+1) - rand(H1,In+1));
        W2_i = coeff_rand*(rand(H2,H1+1) - rand(H2, H1+1));
        W3_i = coeff_rand*(rand(H3,H2+1) - rand(H3,H2+1));
        W4_i = coeff_rand*(rand(Out,H3+1) - rand(Out,H3+1));
        Weights1 = W1_i;
        Weights2 = W2_i;
        Weights3 = W3_i;
        Weights4 = W4_i;
    else
        W1_i = W1_out(:,:, end);
        W2_i = W2_out(:,:, end);
        W3_i = W3_out(:,:, end);
        W4_i = W4_out(:,:, end);
    end

    sim('config5sim_KinematicController_Main.slx');

    % Trailer MSE
        % Position
    xQMSEloss = sum((x_ref - xQ).^2)/t_steps;
    yQMSEloss = sum((y_ref - yQ).^2)/t_steps;
    QMSEloss_pos = xQMSEloss + yQMSEloss;
    QMSEK_pos = cat(2, QMSEK_pos, QMSElossK_pos);
    QMSEKNN_pos = cat(2, QMSEKNN_pos, QMSEloss_pos);

        % Velocities
    vQMSEloss = sum((ref1 - vQ).^2)/t_steps;
    wQMSEloss = sum((ref2 - wQ).^2)/t_steps;
    QMSEloss_vel = vQMSEloss + wQMSEloss;
    QMSEK_vel = cat(2, QMSEK_vel, QMSElossK_vel);
```

```
155     QMSEKNN_vel = cat(2, QMSEKNN_vel, QMSEloss_vel);
156
157     % Robot MSE
158         % Position
159     xPMSEloss = sum((x_ref - xP).^2)/t_steps;
160     yPMSEloss = sum((y_ref - yP).^2)/t_steps;
161     PMSEloss_pos = xPMSEloss + yPMSEloss;
162     PMSEK_pos = cat(2, PMSEK_pos, PMSElossK_pos);
163     PMSEKNN_pos = cat(2, PMSEKNN_pos, PMSEloss_pos);
164
165         % Velocity
166     vPMSEloss = sum((vref - vP).^2)/t_steps;
167     wPMSEloss = sum((wref - wP).^2)/t_steps;
168     PMSEloss_vel = vPMSEloss + wPMSEloss;
169     PMSEK_vel = cat(2, PMSEK_vel, PMSElossK_vel);
170     PMSEKNN_vel = cat(2, PMSEKNN_vel, PMSEloss_vel);
171
172     % Simulated Annealing
173     if i > 3
174         diff_PMSE = PMSEKNN_vel(i)-PMSEKNN_vel(i-1);
175         diff_QMSE = QMSEKNN_vel(i)-QMSEKNN_vel(i-1);
176         if diff_PMSE < 0.05*PMSEKNN_vel(i) && diff_QMSE < 0.05*QMSEKNN_vel(i)
177             W1_out(:,:, end) = W1_out(:,:, end) + 0.1*coeff_arand*rand(size(
                    W1_out, 1), size(W1_out, 2));
178             W2_out(:,:, end) = W2_out(:,:, end) + 0.1*coeff_arand*rand(size(
                    W2_out, 1), size(W2_out, 2));
179             W3_out(:,:, end) = W3_out(:,:, end) + 0.1*coeff_arand*rand(size(
                    W3_out, 1), size(W3_out, 2));
180             W4_out(:,:, end) = W4_out(:,:, end) + 0.1*coeff_arand*rand(size(
                    W4_out, 1), size(W4_out, 2));
181         end
182     end
183
184     XYQMSEKNN = (1/2)*((x_ref - xQ).^2 + (y_ref - yQ).^2);
185     XYPMSEKNN = (1/2)*((x_ref - xP).^2 + (y_ref - yP).^2);
186
187     figure(2)
188     subplot(2, 1, 1)
189     plot(tq, XYQMSEK,'-r')
```

```
    hold on
    plot(tq, XYQMSEKNN,'-b')
    legend('Kinematic Robot', 'K + NN Robot')
    xlabel('Time')
    ylabel('Combined Error (X + Y)')
    title(['Epoch ' num2str(i)])

    subplot(2, 1, 2)
    plot(tq, XYPMSEK,'-r')
    hold on
    plot(tq, XYPMSEKNN,'-b')
    legend('Kinematic Trailer', 'K + NN Trailer')
    xlabel('Time')
    ylabel('Combined Error (X + Y)')
    axis([0 inf 0 0.2])

    disp(i)

    if sum(XYPMSEKNN) < sum(XYPMSEK) && i >10
        sumse = cat(2, sumse, sum(XYPMSEKNN))
         if length(sumse) > 1 && sumse(end) > sumse(end-1)
            figure(3)
            plot(xyref.DATA(:,1),xyref.DATA(:,2),'--black')
            hold on
            plot(xy_axon.DATA(:,1),xy_axon.DATA(:,2),'red')
            hold on
            plot(xy_trolley.DATA(:,1),xy_trolley.DATA(:,2),'green')
            xlabel('x-position')
            ylabel('y-position')
            legend('Reference trajectory', 'Robot trajectory', 'Trailer
               trajectory')

            figure(4)
            subplot(2, 1, 1)
            plot(tq, XYQMSEK,'-r')
            hold on
            plot(tq, XYQMSEKNN,'-b')
            legend('Kinematic Robot', 'K + NN Robot')
            xlabel('Time')
```

```
            ylabel('Combined Error (X + Y)')
            title(['Epoch ' num2str(i)])

            subplot(2, 1, 2)
            plot(tq, XYPMSEK,'-r')
            hold on
            plot(tq, XYPMSEKNN,'-b')
            legend('Kinematic Trailer', 'K + NN Trailer')
            xlabel('Time')
            ylabel('Combined Error (X + Y)')
            axis([0 inf 0 0.2])
            break
        end
    end
end
```